UC Irvine
ICS Technical Reports

Title

Programming languages considered harmful in writing automated software tests

Permalink
https://escholarship.org/uc/item/4mc5n4q3

Authors
Liu, Chang
Richardson, Debra J.

Publication Date
1999-02-28

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Programming Languages Considered Harmful in Writing Automated Software Tests

Chang Liu, Debra J. Richardson

Information & Computer Science,
University of California, Irvine, Irvine, CA 92697-3425, USA

{liu, djr}@ics.uci.edu

SL BAR
Z
699
C3
no. 99-09

Technical Report 99-09

February 28, 1999

Abstract

Although programming languages are widely used for writing automated software test code, we argue that this is a harmful practice for software quality assurance. Programming languages are designed to implement complex algorithms and do not provide a natural mechanism for describing software tests. Software tests consist of sequences of test actions - such as, inputting test input data, checking test outcomes, and recording test results - and not only executing the application-under-test. We dissect a sample software test written in C++ and identify several harmful effects of writing software tests in programming languages.

We believe that the problems we identify are overcome by using a language specifically designed for describing software tests. We briefly describe TestTalk, a test description language under development, and provide a sample software test written in TestTalk.

Keywords:

Software Testing, Test Automation, Software Quality Assurance, Test Description

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

1 Introduction

We believe that test automation is a must for software quality assurance. Thirty years ago, we faced the recognition of the "software crisis", which was followed by the emergence of software engineering [GJM91]. The challenge then was how to build large systems. By now, we have built many large systems, yet a crisis still exists. This time, it is a "software quality crisis". We know how to build large systems, but we do not yet have the technology to assure the quality of the products we build.

There have been many improvements to building large systems so that we can achieve better quality. We still have a very long way to go, however, before we can get good-by-construction products. Until then (and possibly forever), we must rely on intensively testing our software products before we ship them.

Due to the complexity of modern commercial software, the number of tests we have to run is tremendous. To make matters worse, there are also unavoidable problems, such as frequent requirements changes, very high regression testing demands, inadequate time left for testers, etc. We conclude that at this time, test automation is the only solution to meet the challenge of producing high-quality software. In fact, 17 years ago, Adrion, Branstad, and Cherniavsky pointed out that "For all but very small systems, automated tools are required to do an adequate job" [ABC82, page 163].

Automating a software test involves automatically executing the application-under-test, feeding it with test input data, comparing the outcome with expected result, and possibly recording the test result in a database. Programming Languages are a popular, intuitive way for automating software tests. Programming languages such as C++ have virtually unlimited ability and can certainly do the job of automating a software test. Hundreds of thousands of automated software tests have been created in C++ and other programming languages.

There are other reasons why programming languages have become a popular choice for automated software tests. First of all, test automation tools were not around when pioneers in test automation began their expedition. They had to program their own tests. Second, for complex software systems, available test automation tools are not powerful enough to grep all desired information out for testers. Testers or application developers have to write embedded test code, or test hooks, to actively capture information for test drivers. The simplest way to code such test hooks is to do so in the programming language of the application, since this facilitates their interface. Thus, test drivers and automated software tests are frequently written in a programming language.

To be fair, event sequences captured by record-and-replay test tools and scripts are also popular ways for test automation, especially among not-so-sophisticated testers. However, resourceful project teams tend to prefer programming languages since they are much more powerful.

2 Programming Languages considered harmful in writing automated software tests

There are a number of serious fundamental flaws of writing software tests in programming languages. Let us take a very close look at a sample automated software test in C++ and analyze it thoroughly to discover several problems of using a programming language to write this software test.

The program fragment in Figure 1 tests a project management tool. The tool can manage multiple projects simultaneously. When a project information file is opened, each line shows a project property. When the cursor is placed on a line, a small tip window will display related information about this line. For example, if a line says:"QA: Joe Smith", when the cursor is placed on this line, the tip window will display "There should be at least one QA per project.".

```
 1  Bool Test1(ProjectTree * pProjectTree)
 2  {
 3    STRING expectedText =
 4             "There should be at least one QA per project.";
 5
 6    VERIFY(pProjectTree);
 7
 8    IDispatch* pDispProjects = NULL;
 9    RESULT r = pIApp->GetProjects(&pDispProjects);
10    if(FAILED(r))
11      {
12        ReportFailure("Could not GetProjects()\n");
13        return FALSE;
14      }
15
16    long iProjectCount = 0;
17    if(!GetProjectCount(r, &iProjectCount))
18      {
19        return FALSE;
20      }
21    if(iPrjCount != iProjectCount)
22      {
23        ReportFailure(
24          "Added %d projects, but only found %d \n",
25          iPrjCount, iProjectCount);
26      }
27
28    SendKey("&o");
29
30    SendKey("&f");
31    SendKey("&o");
32
33    SendKey("project1.inf");
34    SendKey("&o");
35
36    GotoLine(11);
37    Pause(3);
38
39    HWND hWnd;
40    long lLength;
41    STRING pText;
42.
43    hWnd = GetWndByName("tip");
44    lLength = GetWindowTextLength(hWnd) + sizeof(TCHAR);
45    pText = new TCHAR(lLength);
46    GetWindowText(hWnd, pText, lLength);
47
48    if (strcmp(pText, expectedText) == 0) {
49      ReportSuccess("The information matches expectation\n");
50    } else {
51      ReportFailure("The information does not match expectation\n");
52      return FALSE;
53    }
54
55    delete [] pText;
56    return TRUE;
57  }
```

Figure 1. Sample software test in C++[*]

[*] We are not at liberty to reveal the source of this sample test, but it is excerpted from a real software test written in C++ to test an industrial software package. The code listed here is modified to make it more readable (believe it or not). Code fragments that are not relevant to our discussion have been omitted.

By the time Test1() is called, the project management tool was already launched and multiple projects had been added. Those lines that are not self-explanatory are explained below:

Line 6: "VERIFY" is an assertion. It checks the integration of the data structure "ProjectTree";

Line 9: "GetProjects" returns a pointer to all added projects;

Line 21: iPrjCount contains the number of projects that have already been added;

Line 28: The hot-key to invoke menu entry "Optimize" is "Alt-o" and "&x" is intepreted as "Alt-x" by SendKey() where "x" can be any key on the keyboard;

Line 30-31: The hot-key to invoke menu entry "File" is "Alt-f" and then the hot-key to invoke menu item "Open" is "Alt-o", after which a dialog window will be displayed;

Line 33-34: The string "project1.inf" is sent to the dialog window, followed by "Alt-o", which is the hot-key to invoke button "OK";

Line 36: Move the cursor to line 11 of the pre-prepared file "project1.inf", which is "QA: Joe Smith";

Line 37: Wait 3 seconds for the tip window to show up;

Line 43-46: Get the text inside the tip window;

Line 55: Release the memory that was allocated at line 45.

There are several problems with this test program:

- Test Input Data Buried

    It is not easy to discover the test input data used in this test. Test input data is totally mixed with test driver code and result checking code. Furthermore, the test input data presented here are in their machine representation. For example, there are three 'SendKey("&o")' in Test1() (Lines 38, 31, 34), all of which mean different things. A reader of this test code will have to mentally execute the program to figure out which steps are actually carried out in the test. Better comments might help but will not change the fundamental mixture here. Besides, inacurate or out-of-date comments can only make matters worse.

- Test Oracles Buried

    It is also not easy to tell which results are checked and which are not. One might expect all outputs to be checked against expected results. This is almost alway a false assumption. In this sample test, only the number of added projects is checked, no content data associated with those projects (for example, project name) is verified.

time it is executed, due to the non-determinism of the C++ memory allocation algorithm. This means a tester must use an incorrect version of the application-under-test that reports a wrong tip window message, and call Test1() in a program repeatedly to reveal this defect by testing. It is very likely that by the time the tester discovers this failure, he will be so confident with his test that he will blame the application-under-test for the failure rather than the test program.

Memory-related defaults are known to be common in C++ programming. Programmers are educated to be very careful about memory allocation. Why should a tester who merely wants to write a routine test have to worry about memory?

Another error of this sample is at lines 43-44. The Tester assumed that the name of the tip window is "tip" and GetWndByName("tip") is guaranteed to a window handle "hWnd". However, if the developer changes the window name in a future version, this test program will now fail.

It is possible to improve the situation here by using more comments and adopting better programming style. However, all four fundamental problems listed above remain true as long as programming languages are used directly to write automated software tests.

For experienced programmers, these problems do not seem too challenging - at least, for this sample test. After all, programmers face these problems daily. Imagine, however, you are a tester who is responsible for hundreds of thousands of automated software tests. But one day, most of your tests fail, because the application changes, such as changing the internal window name from "tip" to "message" (refer to line 43). Yet, you are still required to perform your daily regression testing of all of those hundreds of thousands of automated test. What can you do? Not much other than combing all of your test programs and fixing the affected statements one by one.

We are not arguing that a tester should only handle a couple of thousands lines of code like this and more testers should be added to lower the load. Indeed, there are lots of cases where adding testers are very necessary since software testing is often an under-stuffed activity. But not in this case. Although one test might be difficult to implement, the second one is usually very similar and so is every other.

So, what is the solution? Before we discuss this, let us take a look at what an automated software test is made of.

3 What's in a Test?

A software test, or in short, a test, is the set of all test artifacts related to a single test execution. There are many ways of organizing test artifacts. For example, TAOS manages test artifacts as relationships between test cases, test suites, test criteria and test oracles, each of which hold specific complex information [Ric94]. Our interests in this paper are more focused on a single test rather than groups of tests, thus we say that a software test consists of test input data,

expected output, actual output and execution status (Figure 2).

Test input data
Expected output
Actual Output
Execution Status

Figure 2. A software test

An automated software test has several additional components, since there must be mechanisms to execute the application-under-test, feed it with test input data, capture the output, check the output against expected output, and perhaps do bookkeeping tasks. We say that an automated software test consists of a software test (test input data, expected output, actual output and execution status), a test driver, a test oracle, and a bookkeeper (Figure 3). Test driver is the mechanism to execute the application-under-test and feed it with test input data. Test oracle is the mechanism that captures the output (working with the test driver) and checks the actual output against expected output. Bookkeeper is the mechanism that records actual output, execution status, oracle report, etc. (working with both test driver and test oracle) in an information store for future reference.

Test input data	Test driver
Expected output	Test oracle
Actual Output	Bookkeeper
Execution Status	

Figure 3. An automated software test (gray blocks are code)

It is perfectly approporiate to use programming languages to implement test drivers, test oracles and bookkeepers. In fact, without the power of programming languages, it might be impossible to implement these difficult tasks. However, as Adrion, Branstad, & Cherniavsky pointed out, "the testing operation is repetitive in nature, with the same code executed numerous times with different input values" [ABC82]. While it is justifiable to program those test drivers, test oracles, and bookkeepers once, it is clearly not efficient to program them for every test. Furthermore, these components of an executable software test should be separated from the data upon which they operate - that is, the software test itself. By separating these components, we follow the general engineering principle of separation of concerns, thereby fostering reusability of these components, across software tests and even applications under test, as well as fostering other useful "ilities".

One might argue that by employing a better programming style, such as macros or better defined function interfaces, it would be possible to build clear interfaces to test drivers, test oracles, and bookkeepers, thus making the test program easier to write and to read. While this is true, it would not change the mixture of test input data and expected output with the rest of the test program (Figure 4); there is no way, for instance, that testers can avoid programming issues such as memory allocation in the example. Moreover, we believe that test input data and expected output are the core of a test. For a tester to understand a test, test input data and expected output are paramount. Thus, it is beneficial to keep test input data and expected output clearly separated from each other and from the rest of the automated software test (Figure 5). Only in this way can a software tester maintain a huge set of automated software tests and still be reasonably responsive to a quick regression testing requirement.

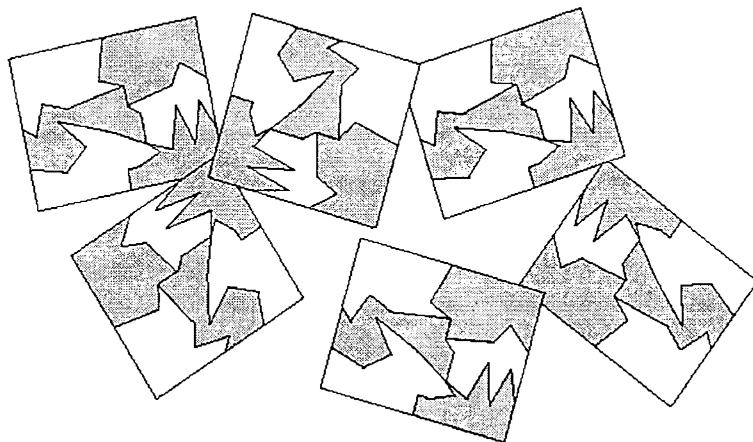

Figure 4. Six automated software tests in programming languages. They are mixtures of test input data, expected outputs, actual outputs & execution status, test drivers, test oracles and bookkeepers. The latter three are in gray, to show that they are actual code. Since one software test might be mixed with another software test, some part of different tests show even more overlay.

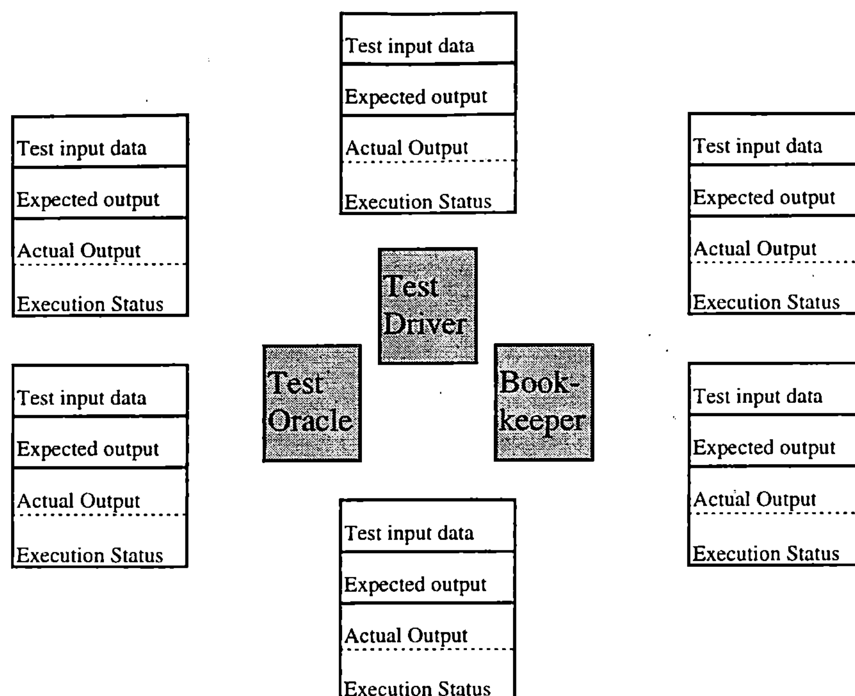

Figure 5. A better way to describe six automated software tests. Three gray boxes stand for test driver, test oracle, and bookkeeper, which are shared among the tests. Test input data, expected output, actual output and execution status of each test are described separately.

A tool that is too powerful is not always a good thing. Arbitrary, undisciplined application of very powerful tools often leads to negative results, as examplified by 1) the Go To statement in programming languages [Dij68] (which lead to structured programming), and 2) open-access to all members of a structure in earlier structural programming (which lead to the introduction of private members of an object in object-oriented programming). Suppose that you need to sharpen a pencil, would you use a knife or a pencil sharpener? A knife is more powerful than a pencil sharpener. But for this particular purpose, a pencil sharpener, which is basically a knife with limited power, does a better job and requires less skill and effort. We think that programming languages are like knifes for writing software tests; pencil sharpeners are needed.

4 TestTalk: A Test Description Language

We are currently working on a software test description language called TestTalk, specifically designed for testers [LR99]. We expect TestTalk to greatly decrease the difficulty of automating software testing while maintaining high clarity of test descriptions. The sample software test we analyzed in the previous section can be describe clearly in TestTalk, as shown in Figure 6.

```
ActionList "Test1"
  CheckWith ProjectCount $iPrjCount
  InvokeMenu "Optimize"
  InvokeMenu "File"
  InvokeMenu "Open"
  Feed "project1.inf"
  InvokeButton "Ok"
  MoveCursor 11 ["There should be at least one QA per project."]
End ActionList
```

Figure 6. Sample software test in TestTalk

Here, test input data (steps) are the main subject, as we would like it. Expected outputs are also clearly marked by explicit test oracle invocation (via the "CheckWith" keyword) or implicit test oracle invocation (checking actual output against expected results inside "[ ]"). Writing a transformation rule set can automate this test. The TestTalk translator automatically generates automated tests in the language chosen by the tester (e.g., C++, a scripting language like Expect/Tcl, or as input to an existing testing tool) when they define the transformation rule set. TestTalk is not our topic in this paper. We include the TestTalk test here just to show that there is a much cleaner way to describe software tests.

TestTalk aims to describe software tests in a manner natural to the software testing process rather than the programming or development process. The goal is to make software tests readable, maintainable, and portable, yet executable. In fact, we believe testers should write tests once, which should then be executable by anyone, anytime, anywhere and on anything.

5 Conclusion

We conclude that programming languages are harmful in writing automated software tests. In an automated software test written in a programming language, test input data and test oracles are deeply buried in the code and mixed with each other as well as automation code. This not only makes it difficult to program such a test but also difficult to debug and test the automated software test itself.

We believe that different parts of a software test should be separated and treated differently. By separating different components of executable software tests, we can achieve better understandability, maintainability and reusability of executable software tests. We are developing a software test description language, whose goal is readable, maintainable, and portable, yet executable software tests.

Programming languages can be used to help automation, but should not be used directly to describe a software test. We point out problems of programming languages in test automation in this paper and hope more researchers can join us in the endeavor of finding better test description languages.

6 References

1. [ABC82] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky, "Verification, Validation, and testing of Computer Software", ACM Computing Surveys, 14(2):159-192, June 1982.
2. [Dij68] E. W. Dijkstra, "Go to Statement Considered Harmful" (letter to the Editor), Communications of the ACM, 11(3):147-148, March 1968.
3. [GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, "Fundamantals of Software Engineering", Prentice Hall, Englewood Cliffs, NJ, 1991.
4. [LR99] Chang Liu and Debra J. Richardson, " TestTalk, A Test Description Language: Write Once, Test by Anyone, Anytime, Anywhere, with Anything", Technical report 99-08, Information & Computer Science, University of California, Irvine, February 1999.
5. [Ric94] Debra J. Richardson, "Taos: Testing with analysis and oracle support", In Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA), Seattle, August 1994. ACM Press.