

AN EXAMPLE OF PRACTICAL COMPONENT TESTING

Catherine V. Stringfellow, Duane Lee York
Department of Computer Science
Midwestern State University
3410 Taft Blvd
Wichita Falls, TX 76308
catherine.stringfellow@mwsu.edu

Abstract

Studying real world software development practices benefits student learning of software engineering concepts. Results of a survey on component testing conducted by a student show how some software developers in industry actually perform component testing and how important they feel certain testing techniques are.

Introduction

Incorporating real world software development practices in students' assignments, projects, file papers, and presentations benefits student learning of software engineering concepts. The concepts become less academic and more practical; the students are not only more motivated to learn these concepts, but also learn about professional attitudes and behaviors of software developers [4]. Data on practices can be obtained from surveys, interviews, collecting metrics on real world software projects, and working with real customers.

This paper describes one example of a student's research for a file paper on software component testing concepts and actual testing practices in an industry setting [5]. His research focused on component testing and the level of importance that software practitioners place on various testing techniques. In an attempt to keep the research practical, data was collected from a survey of software developers working on a project to demonstrate how component testing is performed in an organization. Examples described are based on the experience of a lead engineer on this project.

Background

A *component test* consists of documented set of instructions on how to carry out a particular set of tests on a given software component, and the expected results from the test. *Automated tests* are component tests, possibly generated by a testing tool, that require little effort on the part of the programmer to execute, other than to create an executable for the driver program and then run it. Because in automated testing the steps of the test are in code, the tests are repeatable. *Manual component tests* require some kind of significant programmer action before, or as the test is being executed. When using manual component tests it is critical that the test document exactly what the programmer should be looking for, where to set break points, which variables to examine and at what point, etc. Figure 1 shows an example of a manual component test used by the organization studied. (For automated tests, the documentation is the component test itself.)

Step	Action	Result
1	Start the system using the debugger and set a breakpoint on the first line of the Simulate_Flight.Ballistic_Path procedure.	
2	Initialize system using default parameters.	
3	Transmit target #1 from the database.	Should now be at the breakpoint.
4	Step through each line until you reach the line after Projectile_Location is set and examine its value.	The value calculated should be within 5 meters of the coordinates 510375/3803962/300.
...	... Other steps would be included

Figure 1. Using a debugger to trace execution.

Case Study

The survey data and examples on component testing are from a software system developed by a national company at level 4 of the Capability Maturity Model (CMM). (The professionalism of this organization is especially impressive to students.) The software they develop consists of dozens of independent systems, primarily military in nature. The specific system studied is used for training radar operators. It consists of 91,304 lines of code (LOC), including 72,141 lines of Ada, 1,355 lines of 'C', and 17,808 lines of scripts and data. In general, this is considered a small system for this organization, but provides many good examples for a study on component testing.

A single large component test is difficult to maintain, hence testers create multiple component tests containing related test cases. For the case study system, a decision was made to try to keep the number of component tests low due to the high number of times they get executed during regression testing. Table 1 shows the number and size of the component tests on one of the sub-systems in the case study broken into four high level categories. These categories include:

- Utilities – Reusable components that provide a service to the system that is not specific to this particular system.
- Database & General - Components that provide database services, or services that are system specific and do not fall into any other category, for example a report generator.
- Commo - Communication related components enabling the sub-system to communicate with other systems or devices. Most of the testing is done at the integration and system test level due to the nature of the components.
- User Interface Components – Currently, components in this category are tested at the integration test level and above. This decision to only use integration testing is being re-examined, since there are many cases where component testing would be helpful and because these components make up about 45% of the overall sub-system.

Table 1 indicates the number of components and component test in each category, as well as size measures. The table shows that the mean LOC per component are 224. (This agrees with the survey, where 72% of the programmers said they work with components with less than 500 LOC and only 28% said they worked with components in the range of 500 to 1000 LOC.)

Table 1. Component tests versus components in a subsystem.

Component Category	Components			Component Tests		
	Number	Total LOC	Average LOC	Number	Total LOC	Average LOC
Utilities	26	3,559	137	34	4,157	122
Database & General	31	7,532	243	81	9,349	115
Commo	7	5,381	769	10	869	87
User Interface	72	13,962	194	0	0	0
TOTAL	136	30,434	224	125	14,375	115

For the utilities and general categories, it may seem strange that there is more code generated for the component tests than there is for the component. This is explained by the fact that these component tests require a lot of code, but they are not very complicated to write. The general components have a significantly larger number of component tests. Database and general components are at a higher level than the utilities components and, therefore, take additional code to set up the test

During formal systems test on the radar training system, only 20 valid defects were recorded. Out of these, it was determined that only one of these defects should have been caught during component testing. The remainder of the defects, mainly found in the interface and communication components, should have been caught during integration testing or informal systems testing. This indicates the organization is doing a good job at component testing. Detecting problems during component testing also facilitates testing at higher levels by finding defects earlier in the lifecycle, saving time and money [1]. Table 2 shows information from the organization that demonstrates the importance of component testing, since the later a problem is discovered, the more people, effort and time is required to fix the problem.

The student also conducted a survey to provide data on how component testing actually occurs at one organization in industry. Seven people, who actually perform component testing, were included in the survey. Table 3 provides a summary of the results of the case study survey. Most answers are on a scale of 0 to 10, with 0 meaning no importance and 10 meaning very important. Questions that received no answers, such as average component size above 1000 LOC, are not shown. The survey showed that programmers estimate that they spend an average of 19% of their allocated coding time working on component testing. This included everything from creating the component test plan to documenting the results from running the test. This means that even though a lot of code is generated for component testing, most of the time is spent on coding the actual component.

The survey indicates that the programmers creating the components place a high value on component testing: The answers received ranged from 7 to 10 with a mean of 9. The policy of the case study company is that both black and white box testing must be performed. "White-box or black-box testing improves quality by 40%. Together, they improve quality by 60% [1]." Because of the safety critical nature of the systems, the policy requires a substantial amount of white box testing, even though this is a more costly method. The case study survey showed that programmers are using both forms of testing; however, neither one is viewed more important

Table 2. People, work and effort involved in finding a defect in a testing phase.

Testing Phase	People Involved	Work	Time/Effort
Component	2: programmer responsible for component and team leader	Document and fix the problem; re-execute component test to verify fix	1-4 days (typical)
Integration	4: all the above; a formal tester providing general support; and possibly the group leader	Problem documented at component and system level; fix bug; review/update component test; re-execute component test, re-execute one or more integration tests	3-6 days. Problem only affects the software section at this point.
Informal Systems Testing	6+: all the above and the software and formal testing front line managers. Everyone using the system will also need to be informed.	Verification of the system is same as during the integration testing, but will probably be verified by software and formal testers. Note: There will be much more formal testing involvement at this point. Impacts formal testers since they may not be able to finish executing their test procedures. It may also impact trainers, since they are trying to understand new features.	4-10 days to fix. Probably will take 4-6 months before bug fix makes it into the next informal release.
Formal Systems Testing	10+: programmers, team leader, group leader, formal testers, first line managers and representatives from the Government, User, Independent Testers and trainers.	Meetings(s) held to review the problem, determine severity, and impacts of not fixing it. Meeting is held once the problem is fixed. Additional paperwork is generated. Three different people verify the bug is fixed. Component and integration tests reviewed and re-executed.	2 ½ months assuming fix approval. (6-12 days to fix; 5-10 days to rebuild system; 2 months to re-execute formal tests)
After System Delivery	12+: All of the above plus some users that find the problem may get involved. May impact other users that are using the software.	Formal documentation, two formal meetings, two or more informal meetings, special testing in addition to the normal component and integration tests, special training or documentation for the user about the problem.	Years: Assuming fix approval: 2 ½ months; Extra time to re-field software; Problem may be in system for 2 years (until next formal release).

than the other by the group. When looking at individual surveys, however, some programmers put more trust in black box testing, while others on white box testing.

Test development also follows a lifecycle model of analysis, design, implementation, execution, and evaluation [2]. Company policy states that the programmer who created the component also analyzes, designs and creates the component test. One or more independent programmers inspect it. A lead engineer in the case study project indicated analysis should have identified some test cases for type declarations. In one example, an unchecked conversion was performed when moving data from one structure into another. A new version of the compiler allocated memory

Table 3. Case study survey summary.

Question	Minimum	Maximum	Average
Years of software experience in this field	1	35	12
Importance you put on using:			
Component testing	7	10	9
Black box testing	2	10	7
White box testing	2	10	7
Boundary analysis	5	10	8
Critical points	8	10	9
Normal Data	5	10	9
Random Data	3	10	7
Invalid Data	7	10	9
Estimate of average test cases per unit	1	20	6
Estimate of code time spent on component test	10%	30%	19%
Question	Yes	No	Some
Do you follow the test plan	86%	14%	0%
Do you perform regression testing	71%	29%	0%
Do you document the results of your testing	43%	0%	57%
Question	0-100	101-500	500-1000
Estimate of the average LOC per component	14%	58%	28%

with slight differences, causing one of the structures to change in size so that the data was no longer being copied properly. Another error was due to a difference in how a new compiler compacted arrays.

In the case study organization, design on the component test will typically not start until the component is completed. Advantages of parallel development were found to be less important than time saved by having a relatively stable component when component test design starts. Components are often very volatile while they are being created, and changes to them impact the designs of the component tests.

There is a point in which additional component tests do not improve the reliability of the component, or at least is not worth the additional cost [3]. At the organization in the case study, the decision is left up to the person in charge of the component test, with the exception that the inspection of the component test may reveal additional testing needed. During a “lessons learned” meeting held by the software managers and lead engineers within the organization, it was determined that more components tests be designed for components written in languages that are more prone to errors, for example ‘C’.

Out of the four types of tests (critical points, normal usage data, random data, and invalid data), the survey showed that the programmers felt that all four of these are important. A potential problem with using normal usage checkpoints is that real data is sometimes classified in the case of military systems. A lead engineer at the organization described the necessity of long-term usage tests by giving an example where it would have been helpful in showing a memory leak in a retired system developed by the organization. The prototype system was to demonstrate possible new capabilities for the military radar systems and was used for only a few hours at a time for demonstrations. During one demonstration, the system ran for most of the day due to

delays. Just before it was scheduled to begin, the system crashed for no apparent reason. Fortunately when the system was started back up, it worked fine. Later testing showed the system had a memory leak. A long-term component test was created for one of the components that relied heavily on large amounts of dynamic memory.

For the case study organization, a special program was created to maintain a database of problems found in the systems. (The component test plan also provides a place to record the problems, however this tends not to be a permanent record.) The case study survey shows that 43% of the programmers consistently record problems found, while 57% did not record all problems found. Unfortunately, the guidance on how, when, and where problems are recorded has changed several times at the organization making it difficult to statistically analyze problem reports.

Conclusion

This paper described an example of practical component testing with data obtained from a survey conducted by a student and experiences of a lead engineer on a case study system. This study presented examples of component tests and pointed out the importance of component testing in creating reliable systems in an industry setting.

References

1. Craig, R. and Jaskiel, S., *Systematic Software Testing*, Artech House Publishers, Norwood, MA, 2002.
2. Jones, E, and Chatmon, C., "A Perspective on Teaching Software Testing," *Proc of CCSC: South Central Conference*, Amarillo, TX, 16, (3), Mar 2001, 92-93.
3. Sommerville, I., *Software Engineering*, Pearson Education Limited, Edinburgh Gate, England, 2001.
4. Whittaker, J., "What is Software Testing? And Why is it so Hard?" *IEEE Software*, 17, (1), Jan 2000, 70-79.
5. York, D., "Practical Component Testing with Emphasis on Object Oriented Languages," Masters File Paper, Midwestern State University, August 2003.