# Teaching Software Testing: Lessons Learned

**Janet Drake**
**University of Northern Iowa**
**Cedar Falls, IA 50613-0507**
**Phone (319) 273-5811**
**Email drake@cs.uni.edu**

## Abstract

Software testing is an important and costly part of the software development life cycle. More than 50% of software development budgets are used in validation and verification. This leads to the question -- How much testing do we teach in our computer science curriculum? Over the past five years I have incorporating software testing into my courses at the University of Northern Iowa by: 1) introducing testing concepts in Introduction to Computing, (2) having a 2-week testing section in Software Engineering, 3) and offering a new testing course, Software Testing. This paper describes the benefits and lessons learned in each course. In addition, the author describes her learning experience as a software tester during a summer internship at Rockwell Collins.

Software testing is an important, challenging, and exciting discipline. It is not just a matter of pounding on the keyboard and trying to crash the system. Testing is a highly organized, well designed, and very expensive effort with the goal of ensuring some degree of correctness.

In June of 2002 the National Institute of Standards and Technology released a study [Newman] stating that software failures cost the US economy an estimated $59.5 billion per year. That is 0.6% of the gross domestic product. This points out that we are not testing our software products well. In addition, the report stated that 50% of software development costs are spent on the validation and verification efforts and, on safety-critical software, 66%+ of the cost is spent on validation and verification. As instructors in computer science departments we are preparing people to develop software. If testing is 50% of the effort, we are not properly preparing our students if we do not include software testing in the curriculum.

I have been interested in adding software testing to our curriculum for the past 5 years and have tried several different approaches. In this paper I will describe the 3 different experiences in teaching software testing and their benefits and limitations. The courses in which I have taught testing are:

- Introduction to Computing -- first semester CS students
- Software Engineering -- junior and senior CS students
- Software Testing -- senior and graduate CS students

## 1. Software Testing in the Introduction to Computing Course

Introduction to Computing was a traditional introductory course taught using Ada where students learned sequence, selection, and loop constructs along with subprograms, arrays, records, files, and exception handlers. They completed 13 programming assignments over the semester. Software testing concepts were introduced in the middle of the semester. Equivalence class, boundary value, and error guessing techniques were introduced. Testing exercises were added to 3 of the programming assignments. Students turned in test cases and test results along with their assignments.

### 1.2 Benefits

*Students examined their own work.* As everyone who has taught a beginning programming course knows, students are often happy to just get output. They will not even read the output. This was made clear to me when I had an assignment that calculated gas mileage for a car. Some cars were magic and got over 30,000 miles per gallon. After the testing exercises students understood that they had to look at their output and that many mistakes exist even after the compiler has accepted their code.

*Students retain a general idea of how to select good data for their programs.* I believe that students are left with an intuitive idea of testing that leads them to pick boundary values for testing. I have seen these students again in the Software Engineering course and the boundary value concept is not new for them.

### 1.2 Limitations

*This testing experience is very limited.* Students just get a taste of software testing, but, at this early in their educations, this is about all they can absorb.

*Introductory textbooks do little with testing*. It is up to the instructor to create lectures, examples, and assignments that cover testing concepts.

*For every concept added to a course, another is dropped*. There is only so much time in a course. By adding 3 new testing concepts, I used time that could have been spent introducing other concepts. I believe that the time was well spent in introducing testing concepts.

## 2. Software Testing in the Software Engineering Course

A 2-week section of software testing has always been included in my Software Engineering course. The course introduces the software development life cycle models, analysis, design, testing, teamwork, documentation, CASE tools, software quality, and software metrics. Implementation is not covered because the students are well versed in low-level design and implementation before the Software Engineering course.

White box and black box testing are taught. In white box testing (also called unit testing or structural testing) we are testing the code with the purpose of executing all of the code -- code coverage. We develop a flow graph, and, from the graph, identify paths and write test cases. Our approach results in multi-condition coverage. Students do an exercise where they make a graph, find paths, write test cases, and run the tests.

In black box testing (also called functional testing) equivalence class and boundary value testing are introduced.  I use readings from [Myers] and [Beizer]. The students use these testing techniques as the starting point on a maintenance project. In this project, they find faults, and propose fixes for the faults and improvements in the program. Finally they implement their proposals.

*Note on textbooks and teaching software testing*. General software engineering books such as [Pressman] tend to describe what is easy to teach rather than what is useful and successful in software testing.

| Method | % Faults in programs [Davis] | Book coverage in pages [Pressman] |
|---|---|---|
| Inspection | 65% | 4 |

| | | |
|---|---|---|
| Black Box | 15% | 10 |
| White Box | 10% | 15 |

The reason the textbook coverage does not match the success of the techniques come from the degree of "algorithm" that can be used to teach the subject. White box testing has a nice, clear algorithmic approach. Black box testing has some strait-forward approaches (and some very complex that are out of the scope of general software engineering texts). Inspection is a social approach to finding faults and involves a team of people applying their brain to product. This is not algorithmic and, therefore, not easy to teach.

### 2.2 Benefits

*Students get a testing experience.* They create test cases, run tests, make fault reports, and deliver the resulting documentation.

### 2.3 Limitations

*Students do not spend enough time on testing.* Although it is a testing experience, it is overshadowed by the other parts of the project -- analysis, design, and implementation.

*Students find software development tools difficult to use.* In the Software Engineering course, students use a CASE tool, Visible Analyst Workbench made by Visible Systems Corporation. Although this is a fairly easy to use CASE tool, the students find it difficult. If students are going to do real testing, they will be faced with many more difficult tools.

## 3. Software Testing Course

The 15-week Software Testing course is a project course. Students focus on a major project rather than a series of small assignments. In this class, besides a major project, the students learn testing techniques. Software Engineering is a prerequisite for this course but we cover white box testing again. We use the same approach to white box testing but this time we look at problems with multiple procedures. I expect mastery of the technique.

In black box testing we cover equivalence class, boundary value, and graph based approaches to testing. The graph-based approaches are difficult. Based on the requirements we build graphs that allow us to see and test the paths through the application. We build tests to cover these paths. The graph approaches we cover are:

- Control Flow Testing
- Data Flow Testing
- Transaction Testing

- Finite State Testing
- State Transition Testing

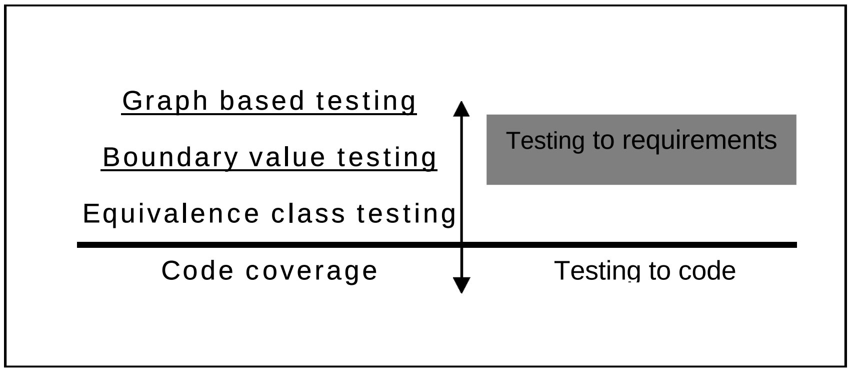

**Figure 1 -- Software Testing Model**

To ensure a reasonable level of testing, the tester must use a variety of testing techniques (see figure 1). For requirements testing it is recommended that a graph base approach be selected. Boundary value and equivalence class testing should always be done. Code coverage insures that all the code has been executed and that no dead code exists in the program.

Equivalence class testing concentrates on inputs. Boundary value testing examines both input and output data values. Graph testing considers the interactions between elements within the program. Together these approaches give a reasonable degree of testing.

Our Software Testing Laboratory is well equipped. We have 5 PCs and a printer. They are not connected to the network for security reasons. Rockwell Collins donated the hardware and the time to install the system. Rational Technologies donated 5 software tools: Rational RequisitePro, Rational ClearCase, Rational ClearQuest, APEX, and Rational Test RealTime. In addition, both Rockwell and Rational have given training sessions to the software testing students.

The material we tested also came from Rockwell and Rational. In the first semester we tested Ada.String -- the strings package for the APEX Ada complier. In the second semester, we tested Rockwell support tools that were built at Rockwell. In addition, we tested a web site, Elementary Citizenship, made by the UNI College of Education for continuing education of practicing teachers. I believe that students have a better experience when they work with software that is or will be used.

### 3.1 Benefits

*Student use a professional testing environment, work on real software, and write real documentation.* This course provides an experience that is close to working in a high-end software development environment.

*Students see the full scope of the testing effort.* They see the creativity and rigor needed to create tests. They see the issues involved in creating an adequate set of tests. They see the documentation effort.

### 3.2 Lessons Learned

*Students must be advanced in their studies to use the testing environment and understand the problems being tested.* As previously mentioned, software engineering students have difficulties with a moderately simple CASE tool. The testing environment is more complex and requires that students have at least mastered CASE tools. The code we tested required understanding computer science concepts.

*Creating equivalence class and boundary value test cases require creativity.* Theese testing approaches are theoretically simple but good implementation requires creative thinking. A good tester sees equivalence classes that a normal human will miss.

*Graph techniques are difficult.* Starting with a requirements specification and creating a graphic model suitable to testing is difficult. The 5 different graphs that we examine are not easy to master.

*A single project cannot give students experience with all aspects of testing*. When the class tested Ada.Strings, they did not face issues such as configuration, compatibility, foreign language, or usability testing. In the second semester a web based project was added so that students could have a wider testing experience.

## 4. The Teacher's Education

In the summer of 2002, I worked at Rockwell Collins as a software tester. I had encouraged the addition of a software testing course to our curriculum and the course was to be offered for the first time in the Fall of 2002. Although I had worked in industry as a developer, I had never worked as a tester. I realized that the job of testing software was more complex than the textbooks indicate. Even in the 1980s when I worked in industry, fully a third of the staff were testers. I was fortunate that Rockwell allowed me to work as a tester so that I could get first-hand knowledge of the complexity of the testing problem.

I worked testing a digital communication package for the KC135 aircraft. The purpose of the digital link between the aircraft and ground stations is to reduce the cockpit crew's radio time. There are many aircraft status reports that can be automatically communicated. I started by running tests. The test engineers had written these tests.

Sometime test engineers choose to write code to execute tests and sometimes they elect to run them by hand. The approach depends upon the number of times the tests will be run.

Through running tests I learned about the application and the testing environment. The testing environment was quite complex. It included documents: specification, test case documents, and fault reports. The specification and test case documents were under configuration control. Each requirement and test case was numbered and traceability was maintained between the requirements and test cases. We used Rational RequsitePro for requirements management and traceability. We ran tests using a simulation tool that simulated both ground communication and aircraft systems for the digital communications software. The simulator was quite a remarkable and complex tool.

After about a month of running tests, I was able to write tests. In addition, I modified existing tests, wrote fault reports, and took part in the Configuration Control Board meetings.

## 4.1 Lessons Learned

*Requirements are the keystone to software development.* I always knew this from my experience as a developer. But as a tester, the requirements are just as important. Functional tests are written to verify that the code meets the requirements. If there are no requirements, there is nothing to verify.

Testers need clear and exact requirements to write a test. We cannot even write a boundary value test for valid data entry if that field is not properly specified. I believe all those who write requirements should start as testers. Once they have had to use requirements to write tests, they know the qualities of a good requirement -- it must be testable!

*The tester finds errors in the specification*. Even though the specifications were well written and used often, they still contained faults. The faults were mainly in the low-level details, but there were still faults and needed to be identified and fixed.

*Creating test cases is very creative work*. We all know that we cannot exhaustively test. There is not time or money in a project to test every possible input value. The tester's job is to pick a reasonable set of test cases that cover the requirements. This takes a high degree of understanding of testing and the domain. A talented and experienced tester can write a reasonably small set of tests that will uncover many faults. A less talented tester can produce lots of test cases -- many more test cases -- but they will uncover fewer faults.

*Testers' creativity is not as easily seen as developers' creativity*. Developers make code and execute programs. We can see talent, creativity, and other good features by simply looking at the code or the executing program. Testers make test cases. It is very difficult to read test cases and see excellence. We cannot see the effort that went into designing test cases. I believe that testers are under-appreciated because their creativity not readily visible.

*Testers know the domain.* Testers see the full application and develop a broad view of the problem. I believe that all people new to a domain should start by running tests. They get to see the whole application and get a feeling for the problem domain.

*Lots of things can be at fault.* When a tester finds a fault, it is not automatically a software fault. Faults can come from:

- Software
- Test case
- Requirement
- Test environment
- Interpretation of results

It is our goal to find the software and requirements faults, and, when we find them we are successful. Just like a developer, we make mistakes when we write our test cases but we don't have a compiler to find them. We find them by inspection and by using them. In addition, when using a complex testing environment, the environment may cause problems. Finally, interpreting results is difficult. What output should we really expect? Determining expected values is one of the most difficult tasks of the tester.

As a tester, I wanted to only valid write fault reports. I don’t want to write fault reports where I later find out that I'm at fault. Just like every other part of software development, testing keeps one humble.

*Fault tracking through the Configuration Control Board (CCB) is complex*. In order to fix a fault, the fault report goes to the CCB and they decide how, where, when, and if the fault will be fixed. It is a complex and labor-intensive operation. At least 4 people had to touch every fault between finding and fixing. The process is necessary to maintain the consistency of the product. When the product is close to delivery, the CCB process becomes intense and the tester is at the center of the process.  In my experience, the fastest fault finding to fix cycle is a week.

*A good team is important*. My team at Rockwell was excellent. The team was centered on the product and the most important thing to each team member was to make the best quality product possible. In my previous experience I also had good teams but our focus differed. I was on a development team and my team's focus was development not the total product. People finding faults in our product were not thanked. I believe that the product centered team approach is more productive and more rewarding for the team members -- especially the testers.

## 4.2 Benefits

*Software Testing Laboratory*. Rockwell Collins and Rational have helped me create the Software Testing Laboratory. Rockwell donated 5 computers and a printer. They installed and help us maintain the system. Rational donated software for the software testing environment. The software includes:

- Rational RequisitePro -- requirements management and tractability tool
- Rational ClearCase -- configuration management tool
- Rational ClearQuest -- fault management database
- Rational Test RealTime -- testing environment
- APEX -- development environment

Both Rockwell and Rational people have given tutorials in the Software Testing course and they also help maintain our complex, integrated environment.

*Continuing relationship with industry*. I am fortunate to have an on-going relationship with Rockwell Collins. Several mentors at Rockwell are very supportive. My classes have visited the Rockwell test facilities and seen demonstrations of extensive testing environments. Rockwell's involvement brings the course closer to a work experience.

## 5. Future Goals in Software Testing

I would like the Software Testing course and laboratory to maintain on-going software testing projects. I would like students to learn testing the same way I did. I would like them to be able to run tests on their first day of class. They would see good test cases and a working environment. They would learn to create test cases later in the semester, but they would have started by seeing and using good test cases.

Currently the students must learn the environment, set up the specification, write their test cases, and finally they get to run the tests. Domain knowledge is also a problem. They have to understand the problem by reading the specification rather than by executing test cases.

## 6. Conclusion

Software testing is expensive. It takes at least 50% of the software development budget and we generally ignore software testing in our computer science programs. It's one of those things that we "hope" is taught in every course but ends up not being taught anywhere. I believe that we can affect the quality of software by teaching our students the principals and importance of software testing.

I believe some testing should be taught early and we should require our students to use those testing techniques in all their courses. I also believe that learning testing requires a mature student and complex software environment. Therefore a senior level software testing course is appropriate.

I was fortunate to have an internship from Rockwell Collins where I learned more about testing. In addition, Rockwell and Rational helped me equip a testing laboratory and continue to support my teaching effort.

Many of our students will start their career in software testing. If they are prepared to test, our students will be successful and be good representatives for their universities.

## References

[Newman] Newman, Micheal, "Software Errors Cost U.S. Economy $59.5 Billion Annually," NIST Assesses Technical Needs of Industry to Improve Software-Testing, June 28, 2002, http://www.nist.gov/public_affairs/releases/n02-10.htm

[Myers] Myers, Glenford, The Art of Software Testing, Wiley, New York, 1979

[Beizer] Beizer, Boris, Black-Box Testing: techniques for Functional Testing of Software and Systems, Wiley, New York, 1995

[Pressman] Pressman, Rodger, Software Engineering: A Practitioner's Approach, Fifth Edition, McGraw Hill, New York, 2001

[Davis] Davis, Allan, Software Requirements: Objects, Functions, & States, Revision, Prentice Hall, Upper Saddle River, New Jersey, 1993

## Acknowledgements

Thanks to Rockwell Collins and Redge Bartholomew in Cedar Rapids, Iowa for their support of our software testing efforts.

Thanks to Rational Software and Joel Pech for their support and software.