Training People to be Software Testers

Jon D. Hagar
10800 N. Sunshine Dr
Littleton Co 80125
303-683-1218
invision@ecentral.com

Abstract
Teaching is at best fun and at worst frustrating hard work, much like software testing. Instructors have been teaching or trying to teach software testing for years. There are books, companies, conferences, and now even college software testing classes. But are we doing a good job? And, how would we know? Which teaching concepts help students learn and understand? I have been teaching practices that receive positive comments from students, though I have no real quantitative proof that they work. I find important elements in teaching are audience (know who you teach to), material flexibility (there is no one size fits all school of how to do software testing), and an appropriate learning model (for example, prepare, present, practice, and perform). From the learning model, I find it is important to have material with hands-on learning experiences, because the big talking head is not how people learn and understand. This paper/presentation focuses on how to teach and not what to teach, as the various schools and possible outlines one can teach from are an out of scope vast subject.

Introduction - Objectives of this paper
What should be a basic goal when offering test training? My goal is to have communication, which results in students understanding aspects of the software testing discipline to the point that they can first *practice* and then *perform* the concepts in the work place. If they can learn and understand, then maybe they will be better testers and will also help in creating better products. What people learn in any class will vary. Some students will need to be given tools for things such as planning or test design, e.g., techniques. Other students will need knowledge of a specific automation tool, such as JUNIT or Rational's suite. But the problem of how to teach effectively remains the same. As teachers, we must understand how people learn, recognize the needs of the audience, know different schools of thought in a subject, and wield different teaching tools.

Background
Software testing is a challenging intellectual activity, which has acquired limited history, methods, practices, approaches, and techniques. Books have been published since Glenford Myers classic work the Art of Software Testing (Myr). Views of how to structure a software test program range from traditional approaches reflected in often criticized documents from IEEE, e.g., software engineering book of knowledge, to recently published works, for example context driven (reference: Lessons Learned in Software Testing - [KBP]) and Agile Software Testing [**LCH].** In addition to books, numerous companies (SQE, QAI, IIST, and others) offer training, as well as a few universities/colleges courses. I have been among those attempting to teach software testing for over a decade. Based on class member's feedback and surveys, my results in training people to be better testers has been mixed. I became interested about two years ago in changing how I taught. I view the workshop on teaching software testing (WSTS) on an extension of this quest.

I teach classes in general testing at the professional and college level. I teach class in:

- Testing embedded software,
- Techniques to ".com" companies, large companies contracting to the government, and small shops looking to survive,
- General software test and quality assurance concepts at the college level,
- General testing concepts to professionals, and
- Test team management.

I view myself as not purely following any particular school of thought in software testing, because I like to be open minded. Besides teaching, I am primarily a test manager and practicing software tester facing real problems in testing every day. I teach largely because I enjoy it.

The audiences for the classes I teach have been varied. I have taught general test classes aimed at the practicing software engineer who needs some common vocabulary and concepts with the testers. There have been classes aimed at "newbie" (junior) testers. I have presented classes aimed at senior staff and managers. Finally, I have taught classes for the college student who is just looking for a general degree in computer science (bachelors or certification) leading to a job. In all classes, I find I get a mix of newbies to the senior experienced engineers, even when the class is specifically stated for introductory level. This complicates classes, and I find I must consider my audience (students) to benefit the range of interests. This turns out to be a hard problem.

The WTST presentation and paper will examine the learning method, and aids, which I try to use to teach aspects of software testing with a variety of students. Further, I would like to focus on where these teaching concepts are inadequate and consider different aids for helping people learn. This includes examining some of the active teaching aids I have used over the years, as well as those that I am developing. These active types of teaching techniques seem to improve learning over the traditional "talking head" lecture in which a teacher follows an outline or syllabus and just lectures. I focus on how to teach and only as necessary what to teach in software testing.

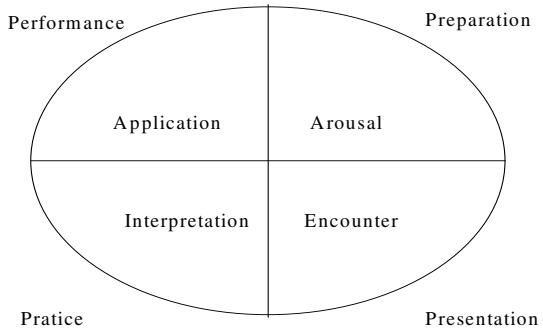
A basic learning model

A key to learning is to get classroom participation and action (doing). This is exemplified in books such as "Accelerated Learning Handbook", "Quantum Learning" and "Telling Ain't Training"[KSK, MEI, DRS]. I am still trying to fully integrate the concepts from such books. A fact I learned when I was a ski instructor is that the best skiers are not necessarily good ski instructors, and having good learning aids (we called them ski tips) can make a good ski instructor. A good teacher:

- Understands how people learn;
- Can provide a demonstration of a method, tool, technique, or concept, such that the student can reproduce the demonstration with practice;
- Has tools, tips, and aids to provide information;
- Can tailor to the audience and student (not every person learns the same way or needs the information organized alike).

While I may at some level "know" these things, putting classes together and implementing the class (teaching it) is hard.

A conceptual model I have used to present classes is based on the Accelerated Learning Handbook [MEI].



This basic model starts in the upper right of the figure with a basic introduction and getting people "excited" about learning (preparation-arousal). This may be hard for a college class where students sometimes think they "must" be there. For professional classes "arousal" is very important in getting people's attention. Next (lower right) is the basic presentation itself where the students encounter the ideas you want them to learn. Effective presentation is a subject by itself (not explored in this paper) and includes ideas about graphics, pictures, audience, style, and concepts. Once the idea presentation has been made, there must be a chance for learning by practice (lower left), where the student attempts to interpret what they have been learning, via implementation (doing it).

Traditional schooling for years has relied on homework and testing as a demonstration of learning. In professional classes homework is likely not possible, so learning must be done in class. I prefer to have in class practice exercises in both college and professional settings. Comments/insights on specific example exercises are defined later in this paper. Finally comes an activity not practiced by many educators (at least in my experience) in which outside of the classroom setting, longer term performance/application is assessed. In lower level general education (high school and four year degree programs), this is difficult. In Master and Ph.D. programs, follow-up between students and teachers is more common. In professional settings, I have encountered classes including application in work place setting. In six-sigma training, students wishing achieve a "belt" level must complete a real world project on the program where they work. Companies establish groups that track green and black belt projects to ensure the students continue their progress. In company specific classes I've taught, I have established long-term, follow-up communications where I can assess changes and "lend" more guidance to students. These contacts often result in return education. This application and follow-up effort is something I intend to expand, since I find it a valued-added educational step.

Expanding on this learning model, I can outline the following important educational lessons learned:

1. To be effective in test training, one must include hands-on exercise. For software testing what exercise is effective?
2. Different audience members in the class require different approaches and have different needs. We need some level of individualism for students in the class (we cannot be factory cookie cutter).
3. There is no one size fits all approach to software testing. Each conceptual approach or so-called schools of software testing likely will have context where it will work.
4. The large talking head (lecture) is a lousy way to learn and understand (see item 1).
5. Our schools of thought, sex, national/ethic origin, personalities, and other factors influence how we teach, but compared to how we may teach, there are alternative ways for people in our audience to learn (see item 2).
6. Learning is often done without words (view graphs) so we need support from pictures, hands on, multi-media, and aids/toys (exercises).
7. Mind and body are separate, and so again to join them, we must have students learn by doing.
8. Learning is creation, not absorbing (see items 6 and 7).

I have experienced these personally, both in skiing and software testing. The paper considers with some detail the ideas of exercises, learning by doing, and individualism (audience – no one size fits all).

I would first like to consider some of the aids and demonstrations I use (with questionable success).

Practice phase: Exercises to understand by doing

I have had and seen many classes where you get lecture. To some extent, I have been guilty of this big talking head syndrome, but from the first class I tried to offer 15 years ago, I have always tried to include classroom exercises and interaction.

So what have you tried? The following table presents some of the "better" learning/practice phase demonstrations I have used. Some of these ideas are noted with credit to the people that initially gave me the idea for exercise. The table also outlines, at a high level, what is learned and how effective I feel it is based on feedback from class evaluations and student comments during the session (not statistically valid). At the end of the table each exercise identifier is expanded with some detail. I would also note, not every exercise as defined in the appendix 1 outline or that I have used is detailed in this table.

Table: Exercises I have tried and found to be effective (maybe)

Exercise Identifier	What is learned	Effectiveness	Comments	Initial idea
Definitions	Terminology	Good	Great way to engage	My

			Class early on. N/A in College classes	paranoia
Test planning Milestones	Plan-based Testing	Mixed – good For traditional	Regulated industries Benefit but Agile Not	Hagar & Others
Error taxonomy/ Lessons learned	History so we Don't repeat	Mixed – People Liked or hated	Formalized error Models seem to upset Some people	Sunday School
Organized Brainstorming	Problems to work In class Problems solving	Good	Can make class seem Or become disorganized	Kepner Tregoe
Learning Exploratory	Exploratory test Concepts	Good though Novice people Struggle on it	Combined in college With plan-based and Other techniques	Bach and Others
Playing with etoys	Can be applied to Many different Techniques	Good – use on Different Techniques	Keep games easy Hands one	E Hend-rickson
Legos	Process efficiency	Unknown	Works only for teams Interested in process	6-sigma
Simple Spread sheet Program	Can be applied to Many different Techniques	Good – use on Different Techniques	Allowed people to test Something they can Understand quickly	Hagar
Shrink wrapped Software	Can be applied to Many different Techniques	Questionable	College class teams Traditional student	Hagar

Notes: TBD

Exercise details

Definitions – Early in a class, students are presented with four or five terms from the world of software testing and asked to define them. Afterwards students are selected to present their definitions of the terms, and most often people use the terms differently. Different meanings can lead to miscommunication and problems in projects because people do not understand each other. The purpose of the exercise is to get people thinking that they should define terms within their project's context and not some global standard. I present notes where I define how I use the terms in the class and ask people to clarify with me when I use a term and don't define it. I tell people definitions in software testing do not always have agreed to, standard meanings. It is not that anyone's particular use of a term is wrong, but we must understand terms are a part of communication.

Test planning milestones – This exercise poses a problem (a set of conditions) and a timeframe. After presentation, students are asked to select from a list of test events and activities to prepare a Gantt type schedule of test activities Students are grouped in teams to create the schedule/plan. After 15-to-30 minutes the student groups each present their solutions and thinking. There are no total "right" answers, but there are some expectations. I used a form of this exercise as part of college tests, but students were not put into groups.

Error taxonomy/Lessons learned from error reporting – A section in the professional classes presents a bug taxonomy and the importance of being able to communicate and understand what a bug is once found. I have tried several bug classification schemes including tables and mind maps. I prefer mind maps. After reviewing "histories" and classifications of errors or bugs in the software, particular to the industry the students are from, students are asked to consider bugs they have found or what has escaped them into the field. The students are asked to generalize errors and attempt to place them in the classification schema. If an error can't be placed, a new generalized classification can be created. Disadvantages of this are:

1) new students (newbie) have no history to pull from, so I do not directly use this exercise with college students; and
2) experienced students have a hard time sometimes understanding how to "generalize" and model specific bugs into a general scheme; and/or

3) Some senior people do not see the value of understanding "history," because they believe the model they use to test to is sufficient, as in people "stuck" in planned-based testing with what they believe to be complete requirements. Some students have argued long and hard about the value of taxonomies. However, students in each class (3 or 4 in every session out of about 20 students), usually comment on evaluations that the error historic and "war stories" which illustrate key points of the class add interest and creditability.

Finally, at the end of the exercise, as part of a reviewing individual lessons learned about bugs, students are asked if errors they have found in the past (but that were not fixed and then later surfaced in the field) might have benefited from more "understanding" and classification (what some authors call bug advocacy). This form of individualization often creates good discussion and more "war stories" (more lessons learned). The disadvantage is that newbies have few stories, and some people do not want to learn from their mistakes (perhaps due to embarrassment).

Organized Brainstorming – This exercise can be done early in a class and accomplishes two things. One is the basic idea of the organized brain storming technique for problem solving. An organized brainstorm is where each person gets to present an idea or problem they wish to see addressed. The teacher usually acts as scribe. Ideas are posted around the room for later referral. In organized brain storming, each person says one idea/problem and only one at time. This continues round robin in class until no new ideas exist or the instructor calls a halt. In the technique, it is okay to pass if a student can't think of anything. This teaches the technique of organized and participatory brain storming which can be of great use to explore test ideas or any engineering problem. The second accomplishment when using this teaching aid is that it helps the teacher identify where to focus the class. Typically, in a professional setting, only a few themes emerge and the class can then be structured to address these. For example, in one class, it became obvious that students knew test planning, but wanted more on Agile and exploratory ideas, which the rest of the class could then be focused on. I found during my early days of teaching (as a ski instructor), that taking each student aside and providing individualized comments helps each person more than group teaching. A disadvantage of customization is many times some of the prepared material must be skipped and "needed" information is not in the standard classroom notes. I have handled this with mixed success (based on class feedback). Some students felt it was "disorganized," while others enjoyed the freedom and the ability to explore new ideas with things like "WOWO board talk" or supplemental notes. There is risk with this exercise and not all cases will work out. This kind of interaction cannot be predefined on charts and likely will be done with WOWO boards. I grew up with teachers who lectured using boards rather than predefined notes, so I like this style and don't find it disorganized, which some people seem to.

Learning Exploratory testing – In recent years in both college and professional classes, I have started combining the plan-based concepts of testing with the ideas of exploratory testing. I have tried class notes and different sequences, but basically, the approach is the following. First, for the "newbie", some basic planning context is presented from the traditional model. In companies where I know the students are familiar with these concepts, I find this is not needed. I then present the concepts of exploratory testing as presented by such people as James Bach. This is immediately followed by a hands-on exercise where students are presented with the etoys problem, simple spreadsheet, or shrink-wrapped software (college). Students are asked to complete 1 or 2 exploratory test cycles. Yellow "sticky" pads are provided to capture each test case idea and assist in ranking priority. These can be posted on the wall for viewing. I also include in the paper/presentation ideas from risk-based testing (part of setting priority). I explain that the way I work as a tester in embedded/regulated environments is in fact a combination of plan-based testing and exploration with risk factors. I like this combination approach because it starts people creating mental models about a product and how to test it. I find mental models to be a key element that most testers have. They have mental models about testing, products, and how the product will be used.

Playing with electronic toys (etoys) – I first learned the use of simple electronic games to serve as the "program" for learning certain test techniques from Elisabeth Hendrickson. Elisabeth applied games to state model-based testing. I have done this and expanded this teaching technique to include exploratory testing, scenarios testing (stories), plan-based testing, pair-testing with exploration, stress testing, and etoys can likely be applied to others. The advantages of these are that they are simple to understand (compared to pure programs like shrink-wrap or open source), and yet they can have fair amounts of test complexity in a short time. There are also interactive and hands-on exercises. Students prepare and execute tests using a

technique that is presented before hand, and then the students present results (bugs?) to the class after about an hour of "play." Students really seem to like the interaction; they refer to class notes for ideas as well asking me about how to apply the technique during the exercise.

Legos – Another toy learning problem, which I learned from the six-sigma training, was to teach "process" improvement with legos. Most organizations have some level of process, either formal (written) or informal (in people's head). Processes that are inefficient are wasteful, but how to recognize them is a problem. Using concepts from state-based model testing, I teach students to use process flows to "map" inefficient processes. We use "yellow stickies" on a wall to represent a process. The yellow stickies can be reordered for process "improvement". The learning exercise with legos starts in a predefined inefficient process to create small lego airplanes. Students actually assemble planes with this process with a 5-minute limit. They then map the process and look for improvement. They then assemble planes with a new process they create by reordering/changing the first process. Times are measured (5 minutes) to see if more planes can be produced. This exercise was practiced in a regulated industry seeking to improve CMM/CMMI rankings. It is not for everyone. I do not teach a particular, detailed end-to-end process for testing (no one size fits all). So, the focus on process here is not prescriptive, but an attempt to get the student to understand that we all end up creating processes when we repeat an activity, and at various time points, we should review personal or group (informal or formal) processes for inefficiency; change them as needed, and then move on. A flow chart/state map, which once done, can be thrown away, is one tool to aid understanding, and by throwing them away one can avoid them becoming a ball and chain--unless somebody wants such things.

Simple spreadsheet program – To teach different test techniques on a computer, several years back, I created a small simple college-grading program in Excel using both "cell" logic, graphics, and Visual Basic macros. Students are presented with the program and told to select different techniques for testing, where the techniques had been in the presentation phase earlier in the class. In one class, techniques presented included pair testing, exploratory, testing, and requirements-based testing (requirements were given within the spreadsheet as a separate worksheet). The spreadsheet program has been seeded with errors, both intentional and unintentional. Students were asked to find and report the errors with the test technique under a timed test cycle (10 to 15 minutes). Students in the class were divided into several teams and each assigned a particular test approach, and then each team was given time to the test program. Bugs were recorded and counted. After each team completed the exercise, the bug report/counts were reviewed. Most techniques revealed similar counts and types of bugs. Each team of students was asked to comment on the technique they used and "compare" notes. Again, I observed that students asked questions and referred to notes as they tried to understand the training of a particular technique. Finally, I would note that I used a variation of this program in a take home mid term given to the college classes. Students were asked to define test plans and techniques by themselves (not in groups) as part of a test question, and then execute tests and find/report errors. This demonstrated numerous areas of test understanding (planning, techniques/design, execution, risk, and error reporting). It counted for a large number of points on the midterm, and so became part of how the college students "learned".

Shrink-wrapped software problem – In the college class, I had more time to let students learn by doing homework. Over a whole semester in the class, the students tested a commercial shrink-wrapped program. The focus was on black box concepts since students did not have code. Other exercises were used for glass box (structural) testing. Students were presented with the program after the first week of class. Student teams were randomly formed of 4-to-5 students. Assignments such as test planning, exploratory testing, techniques and test design, execution, and reporting were all given on the program as the classes progressed over the weeks. Three assignment packages were handed in for grading. Students had several interesting "problems":

- Many (half the class at least) had never worked in group/team engineering class project even though this was a forth year specialized class;
- Team dynamics and grade concerned some students since the team pairing was random, so you had good and bad students together (just like in the work world);
- The ill-defined nature of the shrink-wrapped software caused consternation with some students, again most of these seem not to have ever worked in industry.

This shrink-wrap approach raises several questions, which I have thought of (although more exist).

1) Why did I not select software from the open source community? I did not have a lot of time, and I wanted students to focus on testing vs. learning how to install and operate a piece of open source software. Also many students had been assured programming was needed for this class since it was offered to testers who came from non-programming backgrounds. In future versions of the class, I intend to make some programming experience a requirement.
2) Is it necessary to work in teams? Probably not, but as an industry practitioner, who has hired many new college grads, I feel an area lacking is "teamwork." Classical college education teaches not to share your homework or tests. Yet, in industry, we expect or demand that people work as a team, deal with organization dynamics, etc. In many places I teach, I find groups looking for tools or techniques to solve what are largely management/team issues. I wanted fourth year students and people looking for updated education to learn that testing as I have always practiced it is a team effort.
3) Did you find any errors worth reporting? Most errors and issues noted by students were not reported. Given the current state of company help desks, we would not have any visibility into what happens. Visibility of the error process certainly is an advantage in using open source to teach testing. One team did find an error that could crash the program, and another team found an error where data seemed to be corrupted. Both of which would have been worth an error, had we been of a mind to do this.
4) Did you use different programs? Yes and no. There were 4 teams and three programs; so one program had two teams.
5) How complex were these programs and what kind of programs would make the best candidates? One of the programs was probably too simple as it was a basic "how to" guide. Another program was a drawing/greeting-card creation program, and the graphics in it complicated things a bit. The best (or "buggiest") program that two groups tackled was a "legal form advisor," which required inputs/text processing as well as some boundary value types of testing. In the future, the text input based type of program would seem a good choice as it was not overly complex and had some good input possibilities.

From my days as a ski instructor, the above exercises attempt learning by doing and practice, which is an effective way to teach. What other good "exercises" exist? It would be good to explore this question at WSTS. Should we be looking for a textbook of how to teach the various schools of software testing? It may be too early to have a textbook on software testing, but the need to teach it exists anyway. We need to be training the next generations of testers. These and other exercises and homework assignments are needed.

Combined with a class presentation and possibly a school of testing, these exercises present some "tools" a teacher can use to aid student understanding. But just as no one exercise or school of thought is best in every context (a belief of mine that can be argued), there is no single audience of test students and student needs. As teachers, we need to understand differences in students, and consider changing some of the teaching for the student needs.

Audience

At issue in teaching any class is that of audience, and how they learn and understand. In classes, particular college classes, I find students with no idea what testing is. I also find people that do understand and have been practicing software for some time, but have differing knowledge about software testing. The range of students is from none, to textbook knowledge, to some level of practice, to people with more seniority than I have, to managers of software testers. The problem is a class designed for the first types will not work for the more advanced types and vice versa. I believe the focus of an initial test class--at the college level, must be for the first levels, with a few of the "good ideas" being of benefit for the more advanced. Dealing with this returning student is a problem I am seeing, since the college I am working with (Metro State College of Denver) is seeking to get returning professionals. Returning professionals are looking to update skills/knowledge and/or get a certification to enhance marketability (and job seeking). It must be made clear that the focus of the test class is not for a senior practicing tester, but I need to address both. One idea I have but not yet implemented, designed for the advanced student in the college class, is to offer student projects, where a person of a higher experience level might work on a harder personal project, thus getting the "advanced" knowledge they are looking for. This night perhaps keep the "newbie" interested thus avoiding one of my pet issues--the one size fits all education model.

While the college class has problems with audience, I find the professional class to be even worse at times. Classes in a conference setting, for example STAR, get the widest range of participants, from self proclaimed experts, such as myself, to the new college student just hired into X-Y-Z company to do testing, where the class is their first exposure (what I have been calling a "newbie). I find that even if the class outline included in the advertisements tells exactly the kind of level the class is going to be taught at (introduction, intermediate, or advanced), I get a spread of skills. The classes set within a particular company are only slightly better, since there is uniformity of group and direction. However, you still get a spread of skills and experience.

Table 2 offers some considerations for the issues with audience.

Table 2: Audience

Classification	Types of material	Exercises	Years	Comments
Newbie	Basic planning and exploratory Usually looking for anything	Brainstorm does not Work, struggles with Harder exercises	0-3	Standard Class works
Focused team – with a Company	Most all. Usually looking to survive a short class	Any longer exercises Do not work	1-10	Can tailor Class
General audience in Conference	Largest variation in interest	Short 10 to 20 minute Work best	0-30	Teach to LCD
Skilled work	Usually looking for one idea or specific trick/tool	Likes harder exercise Brainstorm is must	3-30	Results Vary
Expert	Ditto	Ditto	Ditto	Can be Determent

Note:
LCD – lowest common denominator, but organization of material becomes a big issue.

In skiing, we have 3 or 4 basic levels of students: beginner, intermediate, expert, and professional/racer (maybe). When I start my classes now, I do introductions, where I attempt to understand each student. I then use some of the early exercises to refine the understanding, identify the goals of each student, and ask for feedback during the class to redirect our focus as much as possible. Again, for some students, this style appears not to work, while with others it does. I do not want the one size fits all, so I suspect I must accept the few students who do not like the "open" style or message.

Another consideration with audience centers on issues such as race, national origin, background, sex, religion, and student personality. Considering these to any large degree is outside of the scope of this paper. But, the individual characteristics of each student does effect how they learn, and therefore, must be reflected in how we teach. This, again, is the "no one size fits all problem." I am still struggling with these factors, and my first step has been to recognize they exist.

The training literature I reference advocate that people learn best by doing, and I continue to work to address this audience issue. Addressing the issue of audience diversity continues to elude me. Having a person in the class who believe they have the only model of how to test and what I am saying in part or in whole conflicts with their mental model is difficult. These closed minded people can be encountered from any domain (Agile, traditional, programmers, testers, etc.). I have learned to accept these people as well as engage them as best I can, but they can be disruptive to the rest of the audience. And, while I have never done it, I reserve the right to excuse people like this.

The issues of dealing with people in class (audience) become a social science and psychology set of issues, and again argues much large treatments than this paper. With the audience issues, comes material for the presentation.

Material

I find another large number of comments, which I get from student feedback, center on material and organization. I continue working on improvements in outline and content (see outline appendix 1). I never

fully happy with my material and have continuing questions. Should we teach the way to test? Should our teaching cover the multiple school and thoughts on testing (reference [PETT] for "schools of testing")? Can it be that the differences in audience cause some of the problem with organization and what to teach? One organization will make one person or viewpoint happy, and totally disgust another. My experience says the answer to the last question is yes. I do not have good answers to the other questions and they remain in part "open."

We each learn differently. I have hundreds of books related to testing, computers, and technology. My home library occupies four 4 ft x 10 ft bookshelves, and I have hundreds more magazines, reference articles and books in boxes in the basement. Should what we teach be separate from any context and particular "school"? This is probably not possible. But here I have a quandary. As I have learned more, I see value in each school of testing as Pettichord identifies them. I have tried to create frameworks to present different schools of testing. This has lead to practitioners in a particular school of knowledge being indignant about the material from another. While I hate analogies because they represent some level of risk, this strikes me as similar to what has happened through the ages in the art/painting community where new schools of thinking, can earn the scorn of other schools. As a non-artist, I care less about the school and more about the benefit (does the art speak to me and is someone willing to pay for it). I view some of the material I present the same way. I try to speak to what individuals in class need, using different schools of thought as needed. If I succeed, they become a better tester and, hopefully, help their companies create products that people will buy while being good enough to get return customers. While it is disconcerting that I cannot keep everyone happy in my various audiences, intellectually, I know that the material probably cannot make everyone from every school. I think until we have accumulated more history and have better studies of things like "schools" of software testing, it is short sighted to not recognize the current diversity.

Each book, article, and class represents a way someone has thought about the "hard" intellectual problem of testing. Some people are seeking different styles of testing, and I cover as many as time allows. In my material (appendix 1), I present one possible framework for learning different styles. I try not to take sides, as I do like some of the thoughts from the context-driven people, who I might paraphrase as "it depends."

Follow up practical application of what is taught in the work place

A final step I have done, but seen very little of in other classes on software testing, is the post class follow-up. I have done email follow-up, but the kinds of "do it for real" practice seen in the six-sigma training has an appeal, though I do not like the "heavy weight" follow-up aspects I have seen in six sigma. Is there a middle ground? Should I as a teacher plan on a return session (I have done this) as part of the price of the class? How should feedback be gathered (six sigma is very heavy)? Are community forums (like conferences, workshops, and email groups) a good way to continue learning? These are questions that should be explored in WSTS or elsewhere.

Some personal beliefs – subject to debate

I have come to believe that both groups--professional and college students, learn best by doing. The classes I have taught many times where I spend a day trying to stay awake while listening to the teacher lecture, are nearly worthless. Taking a test is one way students learn in college [Kan], but teaching college students with the homework and test method of learning misses things like team interaction and the hands-on aspect of discovering how to break a program while doing testing.

I believe a testing class at the senior college level (or Masters level) must examine both the traditional views of software testing as well as cover the "ugly" reality of much of the industry. Many classical books and industry classes or certifications on testing assume people have such things as time, requirements, understanding developers, or clear support from management. The reality of the work world is that many people find themselves hired for testing jobs with little knowledge of testing, no time for training, minimal budget, questionable products, unrealistic work schedules, and demands from management that they must "be the quality" organization (i.e., be responsible for product quality). A class must address these conditions to help people be prepared for such realities and have some idea of how to deal with them. In addition to preparing students for the reality of much of the work world, there are "traditional" companies, and so the class must also address the textbook and classical skills. We must know our industry's history or be doomed to repeat it. How many papers have been presented at conferences with a new idea (and buzz

phrases), which is basically a classical concept rediscovered or renamed? The mix must be designed to give people the beginnings of testing skills, so they can have immediate impact within the organizations they work for. These organizations expect "bugs" to be found and do not want people just expounding theory. The class training must address this dichotomy. I would like students after my classes to be practical (i.e., be ready to plan, test, and find errors), understand models of testing, as well as be conversant with historic ideals.

The problem is that we do not have a common body of knowledge and models for understanding in either software engineering or testing that is universally accepted. Such documents as the software engineering book of knowledge (SWBOK) are at best preliminary and in need of many years of intellectual discourse before they are usable. At worst it is hopeless dated academic pieces of work, which likely needs to be trashed and started over. Likewise there are several commercial "bodies of knowledge" in testing that are offered by companies selling certifications. I do not believe these should be strictly used as a basis of a class, though the class may want to acknowledge them and provide a student with enough background where after the classes and with a minimal effort, a student could pursue things such as certification. The reason for this is that many companies do base hiring on such certifications and standards. I think working issues of a common body of knowledge and certifications are out of scope for such a workshop. But, the workshop could address creating an education package that, while not solving these long-term issues, acknowledges them and creates a program where students are not "surprised" by such quagmires. And, in the case of certifications, the students can go get one if having it is to their advantage.

Going into the future

I am now working on updates to the Metro State College testing class (for summer of 2004), which is part of an advanced software engineering certification for Metro. This class has expanded traditional outlines and examines ideas from movements such as Agile testing, context-driven testing, or books like "How to Break Software." The first Metro class addressed these, but I am interested in understanding more ideas in the following areas myself, so that future classes can be improved.

1. Traditional textbook learning where we plan and follow the classic processes, such as structural and functional testing to produce a high degree of predictability in software quality. This is traditional test "training" following traditional concepts.

2. Examine Traditional vs. Agile concepts including testing as advocated by people such as James Whittaker or Brian Merrick. I would like a hybrid approach to be possible where both "worlds" could be practiced.

3. How does one start creating a "testers" mind (testers have mental models that they use to do testing)?

4. Examine how changes in testing and SQA help improve quality as a competitive advantage for companies and projects.

5. Testing as an aspect of risk mitigation and development (e.g., test driven design).

6. Failure as a learning tool using bug histories and industry lesson learned to drive test programs. There are no bad tests unless you fail to learn anything from them, and then put those lessons learned to work at your advantage.

7. The time box of testing--control of resources, time, people, tasks and schedules. How does a tester make things efficient, optimal, and good enough for the problem at hand? I have known test/SQA people that only wanted perfection and others that just did not care about anything. I feel that both attitudes are wrong.

8. Short daily planning for testing.

9. Story telling to support testing.

In the workshop, I would like some feedback and consideration of what parts should be integrated with the more traditional outlines seen in most test books and many classes on software. It is not clear to me which model/approach will work to produce good testers in the largest number of students. As teachers, we need to consider that not every approach, concept, example, and class will work for every type of student. The proof of success will be in creating individuals that are interested and excited about being testers and SQA people. And, who in turn help organizations create better products. Follow-up must happen in the form of following students in practice and seeing if their customers and managers see any differences. I have several students I follow year after year. These different approaches are worth considering within the workshop because if as teachers, we are not making better practitioners (coders and testers) and hence better products, then we are fooling ourselves.

Conclusions

Software still has some first and second-generation practitioners in the field who are only 50 or 60 years old. Many of us now teaching went to school before ideas like software engineering or a focus on separate discipline of software testing even existed. Each generation of discipline can learn what the "masters" know, and then push the forefront. This model of master/teacher and student is as old as mankind. What I am attempting to understand is what does it mean to teach software testing to that next generation. It has been said that those that can't do, teach. I believe this is wrong in most cases. The best teachers are those that can do and learn how to teach (master two separate and hard disciplines).

References

[Pett] Four Schools of Software Testing, Bret Pettrichord, Workshop on Teaching Software Testing, Feb 2003
[Kan] Assessment in the Software Testing Course, Cem Kaner, Workshop on Teaching Software Testing, Feb 2003
[Myr] The Art of Software Testing, Glenford Myers, 1979.
[KFN] Testing Computer Software, 2nd Edition, Kaner, Falk, Nguyen.
[LCH] Testing Extreme Programming, 1st Edition, Crispin, House
[Mei] Accelerated Learning Handbook, 1st Edition, Dave Meier
[DRS] Quantum Teaching, 1st Edition, Deporter, Reardon, Singer-Nurie
[HSK] Teaching Ain't Training, 1st Edition, Keeps, Stolovitch
[KBP] Lesson learned in Software Testing, Kaner, Bach, Pettrichord

Appendix 1: Outline of typical testing class I have taught in the past

Top level outline
- Introduction
  - Setting expectations
  - Generate interest - Exercise: Brainstorm
- Definitions
  - Mine are not necessarily yours (no one is wrong)
  - Exercise: Definitions
- Planning
  - Software lifecycles - Agile, iterative, evolutionary, traditional
  - Test planning - approaches and strategies
  - Software test activities - non prescriptive with different flavors
    - Detailed Planning (first pass)
    - Exploration of product/environment
    - Considering key elements: risk, customers, systems engineering
    - Test design (refinement and inputs)
    - Implementation
    - Results analysis
    - Test iteration (more refinement, cases, regression, and alternative activities)
  - Exercise: Planning (if group/company follows plan-based test model) and/or
    - Exploratory with planning
- Test Techniques and levels
  - Developer based testing - Unit, Integration, Functional and structural with
    - coverage of each, Test driven design ideas
  - Classical techniques - boundary value, pair testing, exploratory, etc.
  - Exercises possible: Etoys, Shrink-wrap, and Excel spread sheet

Test automation
- Tools
- Pitfalls
- Model-based testing (embedded class)
  - Exercise: Keyword/data driven model testing
- Decision guide and status quiz
- Exercise (done if organization/company wants automation):
  - work status quiz for your project

Error taxonomy and test reporting (bugs)
- Importance of repeatability and assessing "real impact" (be a good writer)
- Mind map of error patterns
- How to use mind map
- Exercise: mapping your errors to the taxonomy
- Error based testing (tests reflect history) - good and bad (pesticide paradox)

Odd and ends - Verification, validation, reliability, metrics, management, the tester, etc.
- Selected in brain storm
- Backs up charts are included (and if lucky they cover a topic)
- Exercise: some specialized cases exist

Summary

Bio Jon Duncan Hagar.

I am a practicing software tester and test manager with over 20 years of testing experience. I also teach software testing at both an industry and college level. I am a visiting professor at Metropolitan State College of Denver (MSCD) (in Denver, Colorado) where I have taught software engineering and testing classes. I taught a test specific class in 2003 which will be repeated in 2004. I have also developed testing classes and taught software testing within industry for over ten years. My industry experience includes teaching testing to practicing engineers within companies as well as at conferences, such as STAR (East and West), Digital Avionics Systems, and Software Quality User's Group of Denver (SQUAD). Further, I have been mentoring testers in a variety of industrial settings covering both test practices and management. I have written various papers on software testing for publication over the last ten years. I believe I bring both the practical viewpoint of what industry finds important in software testing and the problems facing educators today in teaching this subject. I am neither a consultant nor a pure academic, but believe I am conversant with both areas of the industry.