Principles of Software Testing for Testers

About this course

This course is intended to help customers learn how to test software better. Parallel to the Rational courses, RMUC and OOAD, this course focuses on principles and proven software engineering practices, and does so in the framework of RUP. The course does not require a computer and does not delve into tools.

Most of this course was developed by Cem Kaner, with guidance from Paul Szymkowiak, who wrote the RUP content in this area. The course draws on Kaner's notes from his industrial courses on black box software testing, his academic courses at Florida Institute of Technology and his work (supported by the National Science Foundation) to produce a national core curriculum in software testing.

About these notes

These supplemental notes are intended only to help you, as a Rational instructor, prepare to teach this course. These are not for distribution.

Most of these instructor notes are taken from transcripts of Cem Kaner's delivery of a Master Class 4/6-7/2002 to RUP instructors. When you see the first person singular in the notes, it is a quote from Kaner's presentation.

As compiler, I (Sam Guckenheimer) *have* tried to edit the transcripts for improved readability and to tie the discussion to the appropriate slides. I have *not* done a thorough copy edit. There are lots of transcription mistakes that remain in these notes. (These I have classified as P4 bugs – no plan to fix.)

Only four modules are covered – the ones from the master class. Not all slides have supplemental notes here and the notes vary enormously in length, depending on the breadth of discussion that a topic received in the first master class.

Along with the course notes, every student will get a book, *Lessons Learned in Software Testing* by Cem Kaner, James Bach, and Bret Pettichord. The student notes make frequent reference to this book. To keep the student notes simple, where the content overlaps with *Lessons Learned*, the notes reference the pages or chapter of the book.

Sam Guckenheimer
August 28, 2002

Table of contents

Common controversies

Purpose of this section

The purpose of this section is to help you as the instructor prepare answers to topics that students might raise, sometimes very legitimately, and other times with private agendas. The sections that follow distill the discussion from the master class on 4/6/02.

General classroom management

My responsibility to the room is to close the discussion down when I see only a small portion of people that look to be engaged. It does not mean it is not a valuable discussion, it means that there are some discussions that will be just as valuable with 3 as with the full room. And for those we'll need to take it off-line. The time's that we'll take discussions off-line will be as follows, I am available every day at lunch to meet with 3, or 4, or 5 people to sit around the table and carry the discussion. In fact, we'll have a flip chart and we will note what the lunch time discussions are and we'll pick them on a day to day basis – this is the day we'll talk about this.

I'm also willing to come in ¾ of an hour early for class if at least 3 people will actually sign up and show up to do it to facility a relatively private discussion. Given that, does anyone object if I close the discussion when it's my sense that there are not more than 3 or 4 people actively engaged? I've never had anyone have the nerve to object. As a ground rule for the class, this is something I say this during the time I'm telling them where the bathrooms are. During the opening ground rules for the class, no one has wanted to invest in a fight with me.

And later if there's a discussion I think is going too long, and someone protests when I suggest we close it, I get to say, okay, how many people want to spend 10 more minutes on this. I can't remember a time when I've been surprised by a lot of people saying this. But if the whole class wants to go with this, great. But if only a few put their hand up, you say great, we'll move this to one of the lunch time discussion and that's wonderful. I get pretty nice course reviews from those people who got bored in those things and I shut the discussion down, they say thank you, thank you, thank you. And the people I do shut down, understand it's part of an organized process and it's not personal.

Which is the one technique / style I should use?

(a.ka. If I have an infinite stuff of things to do and a limited time to do them how do I pick which technique to do?)

The course gives perspective on a wide range of testing styles. All of those are valuable to practice in any company. Most companies don't apply all of the techniques; they apply a subset. I've seen some that were very successful and applied 4, 5 at most. But picking which 5 are right for the organization it's a lot better to make that decision based on your knowledge on what the alternatives are and which things are easily automated in a practical way inside your organization.

From my point of view, the more senior a tester gets the wider the range of techniques they understand and the wider the range of relationships among the techniques they understand and to a greater degree they understand what juggling is required for each specific case.

And the second answer is: *Start where you are and expand your scale incrementally by adding a new technique. Give your team six months to master it and then add another in the next round. And repeat.* Nothing else will work anyway.

Get better at what you're doing but broaden yourself. As you broaden yourself you'll discover that you're getting deeper you're going to understand the techniques you know better as well {audio not audible} and over time you'll discover that you know many more things and we'll have more complex test documents, more complex plans but richer ones. There is a distinction between the tester who has ten years experience gaining breadth over ten years and the tester who has twenty six-months strung together over a period of ten years who really has six months experience repeated twenty times. But you will unfortunately often find are people who have been in the business for ten years redoing the project twenty times.

One of the core pieces that distinguishes really skilled, capable, mature testers from junior's is our ability to understand that there are many different attacks that can be effective on the same problem. Many combinations of those can be effective on the same problem. And then to apply judgment to find out which piece would be more efficient today for the situation we are in. The recognition of the diversity of approaches, I've got a lot of different tools, and I want to pull the right one out-of-the-box for today and I know how to use that tool well is something that builds up over a long period of time. We can try to develop education that will make that process faster that's part of the objective of this course but that build up of skill is a gradual thing whether it's in University, in private teaching, or in practice {audio not audible}.

When people don't have varied experience as testers and have locked themselves into a specific paradigm, they have a big career retraining problem that they are going to have to face as soon as the situation changes. .

It's not a lesson than that everybody can learn the first time that they hear it. It's an experience-based lesson. It's hard for somebody to hear that the work they're doing is going to be tougher next week that it was last week. They don't necessarily want to hear that. Oh well.

Can I mix techniques?

Many of the techniques readily, obviously blend together.

So if you go into testing this morning and you say what I want to is really good specification based testing the most important thing for me to think about is the spec and I will do whatever it takes for me to really understand how to compare the program to the spec - you're doing spec-based testing. If you're doing that and creating tests for reuse at the same time, you're still doing spec-based testing. If you're doing that and exploring you're still doing spec-based testing. If you look at the spec and you say well, to tell whether this item is true, I want to analyze every risk associated with this claim, you're still doing spec-based testing.

Now tomorrow maybe you come in and it happens that you still have the spec but you say I want to analyze every individual function that is mentioned in the spec and kicked these individual functions as hard as I can. I don't care where they are in the spec's claims, I'm going to say let me find all the information about printing. I will put it in one list called printing then I will take each subpart in printing and hammer it down in itself and if something is not in the spec but I know it is in printing then I still want to look at printing. Well, you're kind of doing spec educated testing but you're doing function testing.

If your focus is on current learning and testing, you're focus is exploratory no matter what object you're testing.

If your focus is on the many ways it could fail you're doing risk-based testing, which might be done exploratory or might be done spec-based. The real question is where is the primary focus in your mind. All of these techniques end up complementing each other in use. Although some individuals get so caught up in the one dominant way of thinking that they don’t look at the others as things to round themselves out and help them do this one approach.

Automation

Should all tests be automated?

(a.k.a. How can we implement a testing process that is agile, exploratory, constantly brain engaged, with automation tools?)

There are a lot of testing practices that can be automated. There are a lot of programming practices that can be automated but not all of them. Tom DeMarco talks at length about the distinction between programming practices that become routine and then get folded into tools and programming practices that are unique project to project and can never be well measured, and never be well standardized, and never be folded into tools.

We have the same problem in testing. As we come to understand more about testing, we come to an ability to automate more and more of the tasks. *Automated testing is computer- assisted testing. It is not computer-does-it-all testing.* So there's always going to be room for people to be operating by their wits. In fact, as the tools get better, the role of the tester is going to be more and more of someone who has to rely on her own judgment more and more while using the support of increasingly powerful tools to take away work that doesn't need to be done directly by a person.

Exploratory testing by its nature is testing that is not routine and yet testing that we have to do under a lot of circumstances because we don't have routine practices that have any assurance of working efficiently for the problem that the exploratory tester is trying to solve. Some of the things we do in exploratory testing turn out over time to be stereotyped enough that we can convert those into practices and automate them, as we spin off the things that look routine.

Equivalence class testing is a great example. People intuitively understood that boundaries were good places to test. Now test case generators generate boundary conditions, it's a routine thing. Exploratory testers who live their life in a boundary testing world end up generating results that are not necessarily very interesting. Not if the testing was done by the programming staff was any good at all.

So we'll find out that as we spin off the routine things from the exploratory tester, the extent to which the exploratory tester is going to be making one of a kind judgments constantly is going to get greater and what we're doing in this course largely is educating that person's judgment.

Similarly, there's a whole lot of testing that can be done through automated regression tests. We'll see that there are risks to applying automated regression tools in some ways and there are benefits to applying them in some ways. We'll probably have some discussion from several folks here about that and there's some discussion in the course notes. No tool solves all problems. All of Rational's tools solve some problems. No one's tools solve the problems that we're leaving as open problems here.

Can we automate exploratory testing?

The only automation would be to clone intelligence and imagination which we'll probably not be able to do within the next year. An exploratory tester brings to the table a curiosity and some intelligence to put together new paths to travel in the application. There are certain tools that can figure out how to travel every path in the application and all permutations given a certain amount of time, but that's not exploratory testing.

So short of cloning the exploratory tester and their capabilities, the only tools we can really offer are those tools that can assist the exploratory tester. What would those tools be? Documenting the moves they're making so that after the tests they are performing we understand what they did. Or tracking ideas, I may be percolating full with ideas but I need to write them down. I need to try this, I need to try this, I need to try this. In the process of doing a test, four other ideas come. So right now the yellow pad is the best vehicle I have for that. So from an automation standpoint, all I can really do is add a small piece of assistance to that.

What experiences have you had with automated testing?

This is a controversy to the extent that you have people who come into the room who say that either automated testing should be done for everything and that any testing that isn't automated is the stuff for fools. And if you don't think there are people who think that way, just go to the Extreme Testing mailing list.

So the two opposite perspectives are:

1. Everything has to be automated
2. Anybody that does automated testing is wasting their time. You evil tool vendors are overselling something that has really gotten some people into trouble.

Both of these are based on experiences, and an experienced tester might be justified by her experience in drawing either conclusion. It's important to show tolerance and respect for either extreme view (and the many middle grounds), encouraging people to describe the experiences that led them to their view.

Requirements, specs and documentation

Should testers demand specs or requirements documents as a prerequisite to testing?

Let me set your understanding of the context of these questions. Here you are, teaching away, and somebody says, "What, you need to come up with test ideas to figure out how to test a numeric input field? Why, that should be laid out in the specification! You wouldn't want to do any testing without understanding exactly what things need to be covered, and that's all covered in the specification/requirements document. So why are you wasting your time on that kind of thinking?"

It needs to be acknowledged, time limited, and gotten past.

The sabotaging nature of this discussion comes from the group of three people in the back of the room who allege that their company follows a process, an ideal process, they are QA and in their company there are genuine and real specs. They write thorough test plans and everyone else is just subhuman. They'd be happy to teach the class. What they would teach would be an interesting, perhaps non-iterative process, that requires a great deal of work to be done by other groups before they can start.

The response that I give to those folks who are quite persistent sometimes is to say that *testing is done on the basis of whatever information is available and in different companies operating under different risk strategies, different kinds and different amounts of information are available to the testers.*

No test organization has complete information. Some test organizations have adequate information for me (as a lawyer) to base a contract on, but still not complete because there's a great deal about how to use the product and how the product will fail in the field until the product emerges, even if the specification was perfectly written. Whatever information they can get, they should. They should go for the most valuable information they can find to the maximum extent the company will provide.

But if they believe they can block development process by saying they won't be ready to test until information somebody else has to provide is provided to them, the odds are that they will discover at some point that *they are frozen out of much of the development process and put on a schedule that makes it impossible for them*. And maybe in tester Valhalla that they're in at the moment, that's not true. They have the bigger axes to knock everybody else down, that's great. But they're going to find on the next project, that *having skills to deal with incomplete information will be of value* to them.

I don't know what else to say to them. And I think that no class that succeeds in being realistic can assume that you have complete information walking in or that you can get complete information by interacting with the development team.

Do you need to update the project documentation every time you run a test that wasn't directly in the spec?

Cost-benefit trade-off is not an unreasonable approach to thinking about the value of documentation. As with every other aspect of the software development project, we have a finite amount we can do, we have a finite amount of time, we have stakeholder who value some things more than others, and the stakeholder values varies across projects in ways that are project specific. It is the responsibility of the project manager to come up with the right balance of investment to maximize the satisfaction of the stakeholders, which includes preserving the safety of anyone who might interact with the software. Safety sometimes is a big issue and sometimes is not.

I don't think that a general process should give guidance beyond saying do something that makes sense and here are some ways of thinking about what makes sense, like a cost benefit analysis. I don't think we can say it's good to fold these back into specifications or that it's bad. For example, if the rest of the development group is not updating the spec when the software design or limitations change, it might well be a waste of your time to try to update the spec as you discover new constraints or new risks while you test.

Whether you update a development spec, you might or might not update testing specs. Later in this class, we'll talk about the requirements planning that testers might do, to decide what information they should put into testing documentation (and how much maintenance they should do to keep it up to date). In some circumstances, it is essential to update and extend the test documentation regularly. In others, this is entirely inappropriate.

Companies also sometimes roll testing stuff into user documentation, especially in cases where we actually have very intensive customer calls and so it pays to put frequently asked troubleshooting questions into the manual instead of just leaving them in tech support.

One of the things I like to do is create a release kit for testing into tech support. The release kit includes things like here are all of the last tests that we did on the printers we actually tested. Here's what the printouts are, the cover all the boxes, this is what we got from the video cards, this is what we got from the modems, this is what we got from the printers, this is what we got from all of these other kinds of things. And the tech support person is happy to have all of those things so when someone calls up and says, “I have this letter and it's printing oddly in this way.” The tech support person can say, “Gee, I don't recall having heard about that but we did test on that printer.” He opens the binder and flips to the appropriate page and says, “Oh it looks kind of vertically stretched, right?” And the customer says, “Yes.” And the tech support person says, “Yeah, that's our product works with that printer. If that's unacceptable, would you like to refund?”

It might not be the most delightful thing for the test support person to say, you would like to say all our test results were fine but if it didn't work -- fine, you don't have a lot of baloney going back and forth. “Yes this is the way the product works, if it's not acceptable, the product is not acceptable. Let's not waste any more of your time and our time on this. This is how it is. If that's not good -- fine we will deal with it from there.”

We found release kits of value with products that were selling into market space of like a million people. Where many thousands of configuration related calls would come in and the cost of those calls and the cost of trying to troubleshoot those sorts of problems on the phone when you couldn't see the output was enormous. I have been involved in that but it was guided by the cost benefit trade-off. It was very clear that there was a processing cost of testing and creating the release kit and that processing cost saved the company many times that cost instead of wasting tech support time on the phone.

I'm saying that the level of the documentation in any place has to be appropriate to the objectives of the product. And those objectives will vary across products, and across stakeholders, and in my experience, it's just not possible to come up with guidance that works in all cases.

There's a good line in Scott Ambler’s book, Agile Modeling, on the principal of when to update documentation. His rule of thumb is, Update the documentation when the wrong or missing documentation causes pain. So if there's a problem created by not having the right documentation, update the documentation. Otherwise his guidance is, don't take the time to do it.

Staffing

Who should you hire as a tester?

The most common controversy I see is between the viewpoints:

1. All testers should be programmers.
2. No reasonable programmer would ever work as a tester.

So I'm hearing that some other folks are pretty talked out that way. And that debate takes what might be a straightforward discussion and turns it into a long, emotionally based dance. What answer would you give?

Let me introduce you to one way of thinking about requirements for a position you want to fill. KSAO is a fairly standard, unpronounceable short form. Knowledge, Skills, Attributes, Other.

- Knowledge – what have you learned in university. You can discover what someone's knowledge is through factual questions.
- You can discover someone's Skill by having them do tasks.
- Attributes – punctuality, characteristics of the person.
- Other is literally that. Do you have a car? Are you willing to fly?

So, any job description you are going to come up with carries with it specifications for all four of those dimensions. And any job description you are going to come with should be specific enough.

Sometimes what you want this person to do, and they might have the same title, Senior Tester, but sometimes what you want this person to do is be your test architect. The person who's going to come in and handle the automated testing issues for your organization. Well that person has to be a reasonably accomplished programmer who has used the tools at a level that goes beyond mere creation of scripts and has built more than one, preferably more than two test frameworks. Preferably in more than one language and can talk in general terms of benefits and unbenefits of certain structures that will support automated testing. So, we have a lot of knowledge there and ideally you will give that person a skills test. They have to show they know how to design a decent structure that will work in your organization, in your company.

On the other hand, the same job title, Senior Tester, might really mean analyst to work with the domain experts and understand what the customer values are and how to turn those customer values into use cases, combination use cases and scenario tests that are based on complex use cases. That's highly skilled work. That person might need a lot more knowledge of tasks analysis, and human factors. They might need much stronger interpersonal skills.

And the test you might give that person might indeed be to have them interview somebody and come back to you with a job description for the task that person is doing laid out in terms of the subtasks that person has to worry about. That person's ability to interview in terms of skill is there but also ability to interview in terms of do people like this person? There's the ability to gather information from you once. And then there's the ability to win me over if I'm the person being interviewed so that I'd like being interviewed by you every week. That charming personality is what I think of as Attribute.

So as you envision what this person is going to add to your organization, you are envisioning very different people. One of the critical things you should think about in your organization is the issue of diversity. Now diversity certainly includes the diversities our lawyers make us conscious about needing to worry about. The more populations that are represented in an organization, the more ways we can imagine how things can fit. And so that's, different subcultures use products in different ways, read things in different ways, and so forth.

That's one kind of diversity. A different kind of aspect is technical diversity. If everybody on your project team is a programmer then none of your project team has any clue about customer experience. And you will have major holes in what you can find because you have no tax analysts and no way to build scenario tests, not really good scenario tests.

If everybody on your team is a SME, then heaven help you when you try to automate. It won't happen and so as you start thinking about the different roles that your group actually plays in your company, or should play, the different services you provide, you can ask, who is the leader for this service? Who is the champion for this service that we supply? And if your answer is – nobody who is really good at it, you just found out a little block that you will want a senior tester to fit in. That's a different attitude from or a different answer from the – needs 3 years of testing experience, experience with 4 different test tools, etc. – but it's one that I've found much more practical.

Contrary view

My questions are mainly focused around the fact that at Rational Software we have this process called the Rational Unified Process that says, test feeds back into development for future iterations. If you've learned something and are up to the fact that requirements change until the day that you deliver the product and then for that product, that release, those requirements have finally gotten frozen. Up to that point, they've never been frozen.

So, my concern is if testers are so far a field from what we say the product should do, could do, for good reasons perhaps, then not folding the information back into the product, back into the spec, back into the design, then I think we're losing something. We're getting some value out of that effort but it is a onetime value as opposed to a guiding set of information brought in from the test group back to everybody, the documentation folks everybody…

How do you get subject matter expertise?

So in terms of hiring, if I'm a testing group with a common problem, I know as an expert the last place he wants to go is test because that's a stigma.

So it's incumbent upon me as a test manager to grow one internally. So is that what you're saying, if you can't find one, you make one?

If you can't find an expert, you may have to make an expert. But you might be surprised at how many people who have a lot of knowledge of subject domains would be interested in working in a test group. If they are valued for their dollars.

If you decide that you want someone who really understands number devices and then posted an ad that says we're looking for someone with this skill set and these are the kinds of things they would do instead of posting an ad that says bachelors degree in computer science, three years experience programming automated test tools, and knows a little about printers too. You will not find anybody with an ad like that, at least not the subject matter expert for printers. On other hand, if you post ad talking about how interesting printers are and what you might do with them in a test group you might find some folks that understand tons about printers that live in tech support, or who live in hardware companies, or who knows where they live, and say that's an intriguing next step in my career.

So don't assume that people won't come into a test organization if you point to their background and say we'll respect you for what you know. That is a very attractive thing to say to someone.

Is the test group's role QA or testing?

Context

[This question often comes up in the context of someone who is happy to disrupt the class where disrupt the class means – I want to hijack the class and have a discussion on this topic for at least the next hour. We might not get anywhere, but I will have a lot of fun when we do it. So, before I come back to any answers to that person when they bring that stuff up, you were laughing enough that I think each of you have dealt with that person. Can you give me some examples of what that person says to open the discussion or what claims that person makes?]

- My group is not called the testing group, my group is called QA group. We called ourselves that because we all got raises over the people that were just testers. And now we can call ourselves, Quality Assurance Engineers, and, of course, that's twice the salary. So we don't want to be called testers and therefore I don't think what we do is testing, I think we do is quality assurance which is a much higher level function.
- Another common one, is our group had the responsibility to make the ship/no ship decision, the buck stops with us. So when we're not happy with the quality, it doesn't go out the door. So we're more than just testers, we're actually concerned with the quality of the product.
- And that's another claim that people use to say that they are quality assurance but let me try to focus on, what is the derailing question or comment that hijacks the meeting? *Why are you wasting my time teaching only about testing when we should be studying something that's much more than testing. I'm a quality engineer not just a tester. Testers are technicians that get paid a lot less than I do. They're low skill – I have a Bachelor's Degree in Science.* What are some other versions so that people who are going through this who have not taught testing are going to recognize when the rat with the nest pops up and squeaks at them, they can go, oh, a rat's nest, I don't need to go there.

The really enthusiastic hijacker who I decide to come back on, says - I've been in quality assurance groups, I've seen quality assurance groups. I promised I was going to do this and please don't take this personally, but I'm going to ask you some questions and you know I might be a more intimidating questioner than some folks. You tell me that you've been in a quality assurance organization I think that's pretty interesting. *That means it was your organization that figured out what the customer requirements were, is that right?*

Short Answer

Rational Unified Process –indeed, almost every product and service that Rational offers – is about assuring quality. This class is about testing.

Let's look at the Rational University curriculum as a whole and all the pieces that are relevant to QA, etc. We'd love to have you in a class on our requirements management stuff if that's what you do. Maybe you should transfer you to that other class. Yes?

Kaner's Answer

Many groups call themselves Quality Assurance. "Assure quality" means that you can determine quality criteria and have the authority to drive some aspects of the development process in ways that will make sure those quality criteria are met.

These groups may sometimes have quality assurance authority -- maybe they can retrain the programming staff. Maybe they can write the requirements and have them enforced. Maybe they can fire the Vice President. What level of assurance is an interesting question. But it goes beyond mere finding bugs and reporting. And it certainly goes beyond just counting numbers. A Quality Assurance group – adding metrics doesn't make you Quality Assurance – but the point where you can change the company process, development process, support process, hiring practices, training practices, in ways that will align the company more with the goal of achieving certain quality characteristics than against those characteristics, then you are involved in Quality Assurance.

If we were to sit back and ask, what would an organization that actually assured quality do, well maybe they would own the customer requirements. And when I say own, I don't mean get to kibitzing, I mean we're supposed to figure out what the customer needs and therefore what has to be built. After all, if quality is a value to some people, you can't assure quality until you know what the values are. No, that's not your group's role? So you're in charge of tech support budget to make sure customers are happy when problems happen? No? Okay, you're in charge of the training experience the customer, the documentation to make sure communication happens correctly? No, you're just involved in reviewing that stuff. Oh, that's interesting. You hire the programmers and train them to the right coding practices? No? What do you assure? It's an unpleasant thing to do to somebody who I've basically decided to write off and have them sulk in a corner. But there people who will adamantly, repeatedly come in with a political viewpoint and push it. And at a certain point, getting them in a corner and then saying we have a terminology difference.

I think I understand what you mean by QA and the piece we're focusing on in this class is the testing piece. The class was advertised as a testing class. Our issue is that there are many technical skills that are involved in testing that are very important from whatever political place in the organization where they work, there are pieces of information that organizations need about the operation of the product and it's those questions we're focused on here. Going into the broader perspective, there are a lot of places we could go, not all of them necessarily would be fun to talk about, but we need to take those off-line. I've probably done that once without embarrassing a person, maybe twice without embarrassing a person, but if they go further, I take that shot.

So far, I haven't had them come back from there. They sit back and they realize there is a different terminology game that they're not going to win. I'm not likely to get a good reference from that person, but I'm not likely going to get a good reference from that person already. And what I'm trying to do is salvage the rest of the class. Does anybody else have experience in dealing with the persistent person who says, this isn't what's important to be looking at. We should be looking at the overall development process cuz we're really QA instead of just looking at this testing stuff which is just fluff anyway?

Steve Hunt's Answer

Yes, so here's what I'll throw out. My belief is that it's every member of the project teams as part of their role, is QA. So, I'm not just talking about testers, I'm talking about analysts, and developers, and the Project Managers. Everybody is responsible for the quality of the artifacts that they are dealing with, that they're creating, to make sure they are good for the rest of the team.

The primary artifact produced by a tester is a defect report and that should contribute overall to product quality and they have traditionally, some grab it - some say, we are QA and we're testers, we are whatever those two terms mean that's what we are, everything dumps to us. So, we don't have unit testers doing unit testing so therefore they're not assessing or being responsible for the quality of the code they're creating and everything is getting dumped to the back end. Some companies do that by design.

I think that the test group primary responsibility is testing. It is QA to the extent that they need to produce quality artifacts coming out of that effort. I would prefer to see and from organizations I worked where it was effective, software quality group that measured all phases of the software development process. So they came to peer reviews, they looked at test results, they looked at test plans, and they provided feedback to the individual worker, and the Project Manager, and to the customer saying, this is how they're doing against their guidelines for quality. But I don't think that that's the test group…

But does measurement assure the quality because it makes a big difference? When I go into a company I like to see where they're fitting organizationally. Are they fitting under the development group or are they under a separate group? And where those managers are over that, what empowerment do they have to change the process? If you're just a measurement group with no empowerment to change things, it's like the defect inspector at the end of an assembly line at the GM plan where I worked right out of school. The inspectors couldn't fix anything. They had to go back to engineering to change something on the line that was putting the defects into the cars. So is the group's role QA, and if it is and they're not empowered, then they're just frustrated because they cannot change the process, they can just report the problems.

I don't think they are QA beyond the role of everybody having some QA responsibility.

In some organizations I've been in the test group is actually a subset of the QA group and the QA group had that empowerment. But specifics are done by the test group to report back to the QA group.

The Process Improvement Rathole

One of the other possible ways to get into this discussion or one an awfully lot like it, is to say that the testers ought to also be working on improving the process. It follows on to the *we-don't-use-RUP-but-we-should*. That's a huge rat hole that testers really shouldn't go down, but many do. And it might be another way to bring up the same discussion. We ought to start a software process improvement group and try to get everybody on board for that. Especially if it's an in-house class. One guy's been wanting to do it and here's a platform for him to do it.

It's dangerous to get into the discussion of we-should-really-start-a-software-process-improvement-group. There are some people that are really invested in the notion, especially if it's in-house, that they need to take more political control of the company.

I sometimes get called in to teach class where it's explicitly a joint classroom presentation and consultation. Instead of having a three-day class, I have five. And, it's not five days of material, it's three days of let's go through material as a way to structure discussion and then we'll have a group mission study and then on the fifth day a meeting with Project Manager folks to try to reach a compromise with them or an agreement with them. And the discussions in those meetings, when we need to expand our territorial reach, are very emotionally charged discussions. In-house, if my mission is there as a consultant as well as a teacher, I'm responsible for dealing with that. You know, this is let's try to figure out what we need to do.

In-house, I end up asking a lot of questions about what's realistic. I don't give answers for what is realistic, but I may give it a shot by asking questions like, "Has anybody tried this before and what happened? Who are the people in the rest of the organization who support this? What do you think they'd do, are the willing to spend time on this, what's their political power and treat it as a proposal that would have to have general corporate buy-in to succeed, ask who else in the company would buy in?" And if the answer is "No one else in the company would buy in, but I'm really frustrated and because they should. They're evil people, and they're bad, and I wanted to join the post office just so I could shoot them…" Sometimes I get into that and at that point I look at the manager of that group and say, "I need your help to understand how to deal with this."

In other cases, it takes them awhile to get a plan together for who they would have to convince. And then you wind them up and go, "Okay, go convince them, have a nice day." And in some companies they might actually succeed. I don't think that's a path that will help any tester on her career path. Am I right for every person on every career path? Not a chance. If they want to manage their career on that path, then that's their choice. The real question for us in this course is whether is that the focus of this course and whether we can move through that quickly and get that person off in her direction and still willing to either leave the room with a smile or stay in the room.

But understand that part of the process that needs improving is the technical process of finding defects in software which is what we do. It's just like in programming, it's important to understand the requirements development practices and so forth, but at some point it's important to be able to write some really good code. You have to be able to figure out how to break things apart and how to merge things together and what your architecture looks like today and what it will probably look like tomorrow. And those are fundamentally technical issues. They're in an organizational context, but they are fundamentally technical issues. And those have to be done with polish. We're trying to get fundamentally technical issues in this room and those have to be done with polish as well. And those will fit within any in whatever structure you come up with. We can spend a little time on the structure, but fundamentally we need to get beneath that and say, given finite information, what's your best technical strategy for dealing with this. Some people will play with that and for some people that's harder.

Variation: What is the ship-decision role of your test group?

- Final approval?
- Joint sign-off with others?
- Provide end-of-release quality evaluation?
- Provide ongoing input about problems in the product, serving primarily as a technical information provider to management rather than a management function.

Why?

Module 1 - What Testers Should Know About Software Engineering Practices

As you will see, this a reduced set of the materials normally presented in the Best Practices module. The intent is to allow you to *quickly* set context for this course, without triggering the objection, “But I thought I was coming to a *testing* course!”

Be sure to keep your delivery crisp and on point to for a testing audience.

Module 2 - Core Concepts of Software Testing

Slide 2.2

The objectives for this set are simply to look at the foundations for software testing. After this set, after Module 2, we are going to be wandering through the different activities that come up in the Rational Unified Process. But before we can do that, we have to ask questions like, “Well, what is testing? What are test techniques? And how should we think about test design in general?”

Slide 2.4: Functional Testing

Now the course title is *Principles of Software Testing for Testers*. Our main focus is an introduction to Functional Testing. And surprisingly, there are many definitions of the words “functional testing”. Here is the definition we mean. We basically mean black box testing. The initial definition of functional testing said think of a program as a function, you give the program inputs, the program processes the inputs, it gives you outputs. Any aspect of the inputs and the outputs that you could study could be thought of as functional testing. If you treat the program as a function and look at all of its characteristics, you are doing functional testing.

Two things ended up making that definition too broad. First, people wanted to study the inside of the function and how it processes. That’s great but many testers don’t have access to the inside of the program. So I want to separate that off as something that is not part of this course it’s part of the study of what we at Rational call developer testing. The other issue is when you think of functional testing, if you are looking at any attributes of the function, you end up looking at things like security, performance, maintainability, and we have stopped thinking about those as attributes of functional testing.

So in this course, we want to narrow the scope of the class to is anything you can learn about the product without having to look at its code – is black box testing. It doesn’t involve performance testing. I say it doesn’t involve performance testing because there’s an entirely different set of tools and an entirely different set of skills that people should be learning who are performance testing. That’s covered in a separate Rational course. I’ve seen black box testing courses try to stretch to performance but all of them in my experience have failed because they get too broad.

Slide 2.5: Exercise 2.1

If we think we're going testing then I think we are executing a program, running a program, with a bunch of questions about quality. And so another interesting question we can ask is "what is this quality thing anyway?"

Exercise guidelines

- •Count off to form pairs
- •Ask every group to do the exercise
- •Give them 10 minutes or so
- •Collect answers and write on the flipchart

•Transition

- "Now we'll look at what some experts have defined Quality…"

Slide 2.6: How Some Experts Have Defined Quality

It’s interesting that many of the discussions that developers and testers have focus around the notion that the program does or does not conform to a specification. It doesn’t matter if the program works to spec if the spec is wrong. If the spec says that 2+2=5 and the tester writes up a defect report saying I don’t think so, well, 2+2 is not 5, I’m sorry, it just doesn’t work that way. And if you go back to other fields, you look through all the discussions of quality through the recent history of American manufacturing then you find several discussions of quality, none of them, not one of the leading theorists of quality control in the US talks about conformance to spec as an important definer of quality.

Joseph Juran, one of the folks the US sent to Japan right after the second world war to rebuild the Japanese economy and teach them the American way of doing manufacturing. And as we know between Deming and Juran, the Japanese learned a tremendous amount about doing quality control and ended up eating American industries lunch because they did it so well. Juran talks about fitness for use. Specifications are useful as describing what you think you’re trying to build. Ultimately, somebody has to use the thing you are making and if it doesn’t work, it’s not any good. That’s fitness for use.

Phil Crosby talks about conformance with requirements but he doesn’t mean conformance with a piece paper that’s called a requirements document. The piece of paper called a requirements document is an incomplete representation of what it is that the various stakeholders who have to interact with this thing actually need. It’s conformance with the human needs, not the conformance with a piece of paper.

Armand V. Feigenbaum is the founder of the field of total quality management. And talks about quality basically if you boil down what he says is, think of all of the people and all of the different ways they might use this product and ask, is it going to work? If not, it’s a failure.

Slide 2.7: Quality As Satisfiers and Dissatisfiers

Juran came up with a distinction that became a breakthrough in a lot of thinking in how to talk about quality. He talks about satisfiers and dissatisfiers. Satisfiers are things like this feature is really easy to use and I can do everything I want to do with this combination of features. If you read *Consumers Reports* you see checklists, here are all the features these products have and almost all those checklists are lists of satisfiers. You get little footnotes in *Consumers Reports* "oh, it has this feature but we don't like how it does it".

The footnotes are the dissatisfiers. If the satisfiers collectively are the things that would make you recommend the product to a friend, the dissatisfiers collectively are the things that make you wish you'd never bought the thing and make you wonder if it would be ethical to recommend this thing to your worst enemy.

- *Testers tend to focus on dissatisfiers*. Testers tend to think that a product is of a high quality if it minimizes dissatisfiers.
- *Marketing staff tend to focus on satisfiers*. They tend to think a product is of high quality if it has the right collection of satisfiers.

One of the disastrous kinds of meetings that I've been in, and I've been in a bunch of these, and I've been in these as the Project Manager sitting beside the Marketing Manager and as the Documentation Manager sitting beside the Test Manager as well as having been the Test Manager in this thing. But it is fascinating being a witness instead of a participant in the debate and watching the Test Manager and Marketing Manager bang heads like this {beating heads} four weeks before release.

- The Marketing Manager comes in and says, "We've just finished the last set of beta trials and we have to have these features, the competitors just came out and they have these features and we have to have to have these features. It won't be a product if we don't have these features, we've got to have these changes!"
- And the Test Manager says, "We have 300 bugs. They have to be fixed. You're not going to fix those, you are insane."

And they go back and forth, they go features-bugs-features-bugs-features-bugs, and at the end of the meeting no matter who won, who wins is a politically result at that point, no matter who won, both of them walk out and I've had both of them turn to me, Project Manager/Marketing Manager come out and point to the Test Manager behind the Test Manager's back and go "That idiot knows nothing about quality." The Test Manager comes out and points to the Marketing Manager and says, "That idiot knows nothing about quality."

Well what they really know nothing about is how to understand that *every person* in that meeting *understood quality in a different way* and was advocating enthusiastically for a very clear vision of what's needed in the product to make it conform to customer's needs. But different groups in their nature look at different aspects of those needs.

Slide 2.8: Quality Involves Many Stakeholders

There are a lot of stakeholders on any project.

Slide 2.9: Exercise 2.2: Quality Has Many Stakeholders (1/2)

In-house class

If this is an in-house class, by now you would have started, probably in the first session, understanding what products the company makes, and how many project teams there are in the company that are represented here. In the ideal case, there are only two or three project teams and so you might split them and say, Look, let's work on one or two products right now.

Open enrollment class

This is a harder exercise in a public place but what is useful to do at this point as an exercise is to get people to try to think through different angles on what quality would feel like. My public exercise at this point is one that is focused on a programmer who joins your group and has these characteristics. The group starts like this.

Imagine that you are a tester in a company where the programmers had all agreed on a set variable of naming conventions. Variable naming conventions are important because if everybody understands how variables are named, than anyone can read anybody else's code and gleen something just by learning what the variables are. In the old days, we didn't have variable naming conventions, I started programming in the late 1960's and it was common in those days to identify the type of variable by its first letter. For example, you might name your integer variables, Irene, Jane, and Karen after your three most recent girlfriends. Totally reasonable, I J K, these are all integers that's all you need to do to specify what they are. And they can only be eight characters anyway so you didn't have the meaning you can have with 32 or more character names like we have now.

So here's this group, they've come up with their list of variable name rules, they hire a senior programmer from generations ago to now. This person is called experienced, this person comes in, he starts writing code, and he starts naming his variables after all the people he's never liked. The rest of the programming are appalled and say *you can't name your variables that way*. He says, *yes I can -- watch me*.

So one of them decides to report into the bug tracking system a change request. The change request says, *We have a variable naming convention standard here. Our standard says variables will be named in the following way that identifies the type and use of the thing and the place where the variable is first defined so Irene doesn't cut it. All of these variable names have to be changed.*

Here's the question I want to ask. ***Does that belong in the bug tracking system? What do you think?***

[Spark discussion.]

Well I want you to play a role. I'd like you to play the role of the Marketing Manager who is going to skim the bug tracking system for information about aspects of the product that might make it less marketable. Why don't you be the Documentation Manager? What do you care about?

You're the Tech Support Manager. You're going to use the bug tracking system too, right? Would you care about seeing this change request? How come?

> *So if one of the quality characteristics for a product is supportability, and if variable names have anything to do with supportability, then if he doesn't see information like that, he's crippled.*

For saleability, it doesn't matter. For end user description it doesn't matter. For support, it matters vitally. You are a programmer, not the one who wrote Irene, Jane, and Karen, do you want that tracked officially? Do you think that kind of thing is appropriate in a change request system?

If issues about defect counts arise, where students want to keep bug counts down, then:

> So we have an interesting conflict, which we'll come back to later, between metrics derived from the database and other purposes of the database.

Every problem report or change request that goes into the bug tracking system, is a demand for a management decision on the record: *Does this meet our corporate quality standards or not?*

Every different stakeholder in the company has a different subset of quality standards. The Marketing Manager is going to worry about some issues, the Documentation Manager is going to worry about some issues, the Tech Support Manager is going to worry about some issues. All of them have different interests, different worries, about what things to track in the quality of the product. But anything that goes in, goes in as a statement that the development team as a whole or the manager of the development team is being required to decide whether this meets corporate quality standards or not.

Now as a tester, you have an interesting situation because if you talk to different stakeholders in the company you're going to have very different visions of what is acceptable and what is not acceptable. The Marketing Manager might easily look at a dispute over variable names and wish that the programmers would grow up and start making features. Whereas the Tech Support Manager might feel like this is a vital discussion and development should stop until this is dealt with.

As testers, you cannot and should not make the decision about which one of those stakeholders is right. That's not our job. The thing we can do is help the person writing the report express their perspective in a change request clearly enough that the rest of the development team can go "Ah, whether I agree with this perspective or not I understand what it is and I can see why this stakeholder thinks this is important and I can understand how important this stakeholder thinks that it is."

There are many different people who have write access to a change request system. The test group is typically the group who helps people improve their change requests in a way that will make them the most effective requests they can be and if we start drawing judgments from our little piece, is this a dissatisfier to a customer, than we are missing all the other dimensions of fitness for use that all the other stakeholders in the company are going to see.

Slide 2.10: Exercise 2.2: Quality Has Many Stakeholders (2/2)

There are some other examples that you might work as exercises in the class. We worked the first one. I hope it's obvious how you'd handle those.

In different groups, if this is a group that's making products that translate everywhere then they're going to find the localization issue much more compelling than the programming issue.

In an in-house class, it's worth finding out what their quality controversies are and having them play the roles of Product Manager and so forth for that product. Then ask them, What have been the most controversial three bugs that have gone into the bug tracking system over the last three years, over the last year? Why should that particular one have been there?

Don't be bound by these slides.

Slide 2.11: A Working Definition of Quality

We’re going to adopt a working definition of quality for this course. That quality is value to some person, different people who have different notions of what is important for the product. But if you’d look at product and say, “This is less valuable than it should be because this decision was made,” then what you are saying is this product has lower quality and unnecessarily lower quality than you think it should have.

Slide 2.12: Change Requests and Quality

As soon as you can attach either an emotional value or an economic value to a decision you're talking about a quality value. From that point, we can think about the word "defect" in a little broader way than failure to conform to a specification.

I think of the word defect as something that is qualitative, something that happens in the eyes of one of the stakeholders. The TSM and some of the programmers thought that Irene for a variable name was a defect. The Marketing Manager didn't think it was because it has no impact on the customer experience and it has no impact on the current salability of the product.

I want to suggest that a defect in the eyes of some person who thinks about quality is something that unnecessarily lowers the value of the product to that person. They look at it and say, *They didn't have to make that decision. That decision really reduced the value of the product in my eyes. I think they should reverse it and approve the product.*

That's a defect report. Several folks call those bug reports but the distinction between a "bug" and a "defect". In court "defect" is often taken as an admission that something is truly wrong with the product in a way that might be legally enforceable whereas bug is typically taken as a statement "well, we didn't like something about the product". So many companies use the more neutral words "anomaly", "problem", "bug", or "desired change" instead of the word "defect" which has connotations that not everybody will like to see in their database. Here we're going to talk about change requests and defects most often because that is what is most often used in the Rational Unified Process.

We're going to hit In Module 6 on what testers should report. The test group is, of course, not the only source of change requests. If someone else comes in with a change request, they're saying they're not happy with the product and if they're willing to sign it, it's their political capital that's being spent, not yours. The constructive thing you can do with that person is to help them word their report and troubleshoot their report well enough so that everybody else on the team will understand why they are spending that political capital and raising the issue.

The tester might not be willing to write under her own name, "I don't like variable names like Irene." But if one of the programmers comes in says, "I think we need to have this in the defect tracking system." In most companies that is the programmer's right. It would be inappropriate for the tester to say, "I don't think people want to see that. I'm not going to let you put it in." It might be much more appropriate for a tester who sees a change request that says, "Irene is an improper variable name."

It might be very helpful for the tester to walk up to the person who wrote that report and say, "I'm not sure that's enough information for people to make a reasonable decision about this report. Why don't you like Irene? What is the problem? What are the consequences for the project associated with the problem? Why don't we put those all together in something that would take the Marketing Manager for example and make him understand the big deal." Every stakeholder has the right to report a defect but many stakeholders don't understand how to put their perspective down on a piece of paper in a way that will convince the other stakeholders it was worth reading. A big part of our task in quality improvement assistance is to help people get that communication.

Slide 2.13: Dimensions of Quality: FURPS

So we can think a lot about dimensions of quality. A traditional way to think about dimensions of quality is being called FURPS. You can think about a product in terms of its functionality, its usability, its reliability, its supportability, and performance. There are some challenges in applying the FURPS notion. FURPS is a very good classification system . If somebody starts talking about something you can say, “I think you’re talking about the functionality of the product” so folks will sometimes keep statistics based on FURPS and that’s useful.

Slide 2.14: It May be Useful to List More Dimensions

But sometimes you need to get underneath these five "ilities" to a broader list of "ilities" in order to generate tests based it.

Imagine a quality risk of the form, it doesn't have the functionality that I want. Well, how would you test for that. Maybe that's too broad. On the other hand, suppose we that we said an aspect of functionality is that it doesn't have enough accessibility.

Well, what does accessibility mean? Accessibility as an attribute of software means that the software has been designed to be useable by someone with a certain handicap. The handicap might be that they're deaf. The handicap might be they're colorblind. The handicap might be that they only have one hand. So the product can be accessible to one subpopulation but not another. If you want to test the accessibility characteristics of a product, then you are going to start thinking about, "Hmm, which subpopulations are there, are the contrast on the screen high enough, do they every have to click the mouse and press the space bar at the same time, do they have to hear things or are there also visual cues.

If you were to think about accessibility you could come up with a list of issues that might not immediately come to you if you thought about the broader attribute or broader characteristic of functionality. When we try to classify what quality characteristics are in general, FURPS works fairly well. When we try to use generators you are well off to ask yourself in your company, what are the main things that are quality related categories or issues for your products. And then start driving things from that lower level. Localizability is another class that some people remember but some people forget. If it doesn't show up on your checklist to think about localizability, it will never happen.

Localizability of the product is the set of features associated with that product that make it easy to localize that product to another culture. An example of localization is to translate all the text from German to English, English to Japanese, and so forth.

But I've been involved in localizing products from American to British, and while it‘s true there were some spelling changes from American to British, many of the changes had nothing to do with the text. It had to do with replacing the flag – you don't necessarily want to have the American flag waving everywhere if you're selling a product outside the US. They had to do with words that were used to express certain concepts – not just translations, how do we display dates, how do we hyphenate, not just translations but a wide range of other things that the people in the place that you're localizing for expect about your product.

So for every one of these and I can guarantee you'll think of others. I've been reviewing a list that a Masters student at Florida Institute of Technology has been generating and this student at this point has 65 quality categories that he's generating test cases out of – it's an enormous list – whether he'll end up bringing that under the FURPS classification or some other one, at some point he'll have to get a hierarchy. But as a generator of test ideas, the 65 categories are actually proving remarkably effective. He's coming up with hundreds of ideas for "how will we test a shopping cart?"

When you take an experienced tester and put him against some technology they've never gone to, they end up asking "Okay, how would I analyze this thing for accessibility, how would I analyze this thing for maintainability?" They come up with a lot of specific questions that they wouldn't come up with from a simple classification system. So coming up with a list like this for your company could be a useful tool.

Slide 2.16: Test Ideas

I mentioned test ideas when I talked about generating ideas for testing a shopping cart.

A test idea isn’t yet a test case. A test idea is a description of what I think it is of what I want to test with a little bit of a notion of how I want to test it. We don’t yet have the details. You can generate tons and tons of test ideas and we’re going to go through an exercise where we do. One of the problems that people are going to have to face and you’re going to see this as we generate for the very, very simple data input field, is an explosion of good ideas to test. You can’t test them all. You don’t have enough time, the critical problem with testing is that we have an infinite amount of work for any non-trivial program we have a virtually infinite population of test cases we could run. Of those, many of them are desirable and more of them are desirable than you will have time to run and so we fundamentally have to trade off all through the process.

All through the development process, we’re constantly going to be asking the question “Are we doing the right thing for now? Are we coming up with the most important test cases? Are we spending our time in the right ways? Every iteration we should ask that question another time.

The fact that we can come up with great ideas for testing doesn’t mean we should run all those tests, we can’t afford too. But having a good population of great ideas especially ones that have been made in advance so you don’t have to spend the time, gives us a chance to select.

Slide 2.17: Exercise 2.3: Brainstorm Test Ideas (1/2)

We’re doing something called brainstorming. I just want to lay out a couple of ground rules for brainstorming before we go back. The goal of brainstorming session is to generate a lot of really good ideas and we’re going to end up as we do this generating some ideas that can be very well refined but start out as pretty rough.

The most effective way to brainstorm instead of thinking about this one is better than that but is to say here’s another one. People are going to come up with wild guesses the first time through and I’d like to make sure that everybody in this room feels very comfortable saying something that, well, it’s the first thing that comes to my mind in this class, we get the class that way. The other thing we get is the example.

Tomorrow, if we were generating a test ideas list, tomorrow what you’re going to have is a set of flip chart notes and vague memory and if you have specific examples so when you see this thing, real numbers, you go, “Oh, now I remember what we were talking about” they meant this. So the specific example is useful not as the best case but as the reminder of the class.

Every one of these will be able to come up with a better example for the class and we’ll probably do that refining off-line. One person will take the list and come up with the best list after that. BTW, I’ll come to your question in a second.

It’s okay to make jokes in a brainstorming session. Brainstorming sessions are allowed to be fun. If you have something that strikes you as funny in the middle of one of these sessions, you’re probably hitting some hidden truth. Irrelevancies fall flat as jokes. So if you come up with something that is funny to you, it’s probably also going to trigger someone else to come up with a relevant idea that may be rather powerful.

Slide 2.18: Exercise 2.3: Brainstorm Test Ideas (2/2)

So here's our test idea exercise. Imagine that we have a data input field. It will take numbers between 20 and 50. What would you test? I want you to give me two pieces of information.

1. I want you to suggest a test case.
2. I want you to tell me what's in your mind when you're thinking of that test case.

Draw two columns on the flip chart: ***<u>Test</u>*** *and* ***<u>Reason</u>***

Sample discussion

Non-integers, Okay, well, I'm going to take that as more of a description of a group of them. Give me an example of what you would test.

Okay, so we've got real numbers as well as a subclass. I couldn't type at a keyboard a real number. If you told me to type a real number I'd start by going R E A L and that's probably not what you mean. What would you type, give me an example. 20.1 – why is that a good real number?

Okay, it's in range and everything is valid about it except it has decimal point. Now, are there any other non-integers besides non-integers that have decimal points?

So a huge number is another one. And an alpha character. Which alpha character would you use? Okay, what else?

If that came up we'd make a decision so what's your definition of a long integer? Okay, that's going to vary across systems. What I'm going to suggest is that we think about an integer as having a maximum as its maximum value and the odds are good that in any integer that we define, 20-50 is within the range of valid values for that integer so max-n is probably out of range for that. Now let's come back to generating a list here.

So I'm writing down all valid values, 20 and 50 upper and lower bound. What else?

020 What's special about 020?

0 Space. So, I'm going to capture the 0 space.

What's interesting about 0? Murphy supplies a lot of laws for testing. One of Murphy's Laws for testing is if you can sneak a 0 somewhere, someone can divide by it. So, it's not supposed to accept 0 but that doesn't mean it won't. And now we have space, I draw a space character as a little half box.

What's interesting about space? Ah, what's interesting about deleting space?

Good, 19 – 51 the other boundaries.

I'm going to stop the brainstorming. But I am going to say, this is a method people use to generate test ideas all the time. A lot of independent test labs will generate standardized test idea lists and a standard test idea list is one you are going to reuse from product to product, to field to field.

Slide 2.20: A Test Ideas List for Integer-Input tests

If you want to test numeric input fields, do you really want to do this reasoning every time you get a numeric input field? Of course not. So what is useful to have is a standard set of test ideas for any numeric input field that is integer, any numeric input field that is float, any one character field, multi-character field. You will not find those published.

The brainstorming process that folks go through in developing these is a three-phase process. The first phase is something everybody does at home alone, tell people in advance a group of 4-5 other testers think about this for 15 minutes overnight, sleep on it. We're going to look tomorrow at this kind of a field or this kind of a problem. Please come up with a standard set of test cases, bring some notes, whatever you can scribble in 15 minutes but make those scribbles the night before not the morning before. Let yourself come up one or two other creative ideas while you are sleeping.

The discussion that happens runs between often 1 hour to 3 hours, the less prepared the students are the longer it takes. It starts out with people with reading off the test ideas from their notes. All these are test ideas. People start off reading those ideas they came in with and once they get those ideas done, you have an interesting silence.

Inexperienced facilitators of brainstorming stop at the silence. That is when the group is just starting to get their work done. Count silently to yourself for 10 seconds. Most of the time the pause will last less than 10 seconds. Your most powerful tool as facilitator is your ability to control your own mouth. Make silent eye contact with every person. Raise your eyebrows. And you will discover, after 15 seconds somebody will come up with something. And somebody will come up with something after that.

You now have the value of the meeting, the people have laid out the foundation for the thinking, they're now thinking together and bouncing ideas off each other. Up to this point it was dominated by people coming up with the ideas they had now they're coming up with the ideas no one could think of in the first few minutes. That's why we want to have people working together.

Let the group keep going until you hit another 10 seconds of silence. Wait, if you wait you will be rewarded. People will come up with yet more ideas. In my experience, people will come up with creative ideas in that last group. Let them go for another 10 seconds and at the end of the 3rd group of 10 seconds some folks are finally getting very impatient. There's no point in fighting that as a group.

As we do this brainstorming they get more mature and less impatient because they realize what's happening as you point it out to them, but you point it out to them at the end of the session. At the end of the 3rd group of 10 second silence, you say, "Okay, let's go for 5 more than we'll stop." Just like someone running a race, see the finish line speed up giving their fastest pace. It's stunning what some of the ideas are that come in at the last five to ten pieces come in as people do that one burst of thinking and then walk away exhausted.

Now that gives you pages and pages like this. I facilitated a meeting recently where we did one character wide character field and we had 15 charts. Does that mean we're going to use them all in a test ideas list? No, some of the ideas were wild and crazy ideas. Some of those led to some really interesting stuff. We ended up at the end of the first day with 14 charts.

Another person who was responsible for that. I was merely the facilitator, the person saying, "Okay you're next." I was not the person responsible for taking that and turning it into a good test idea list for a single-character character field. She took those, wrote the why a little more coherently and also refined the ideas.

The refinement's important. Let's think for example of the alpha character. Alpha characters are an example of non-numeric characters. If we think about how we tell the difference between numeric characters and non-numeric characters, for someone thinking from a purely black box point of view, A is a letter, it is one of the extremes of the letters, its ASCII code is closer to the ASCII code for the numbers than any other letter.

But if you were to look at how that number is processed, there's probably a routine whose name is probably IsDigit and that routine probably says something like: "If the ASCII code for the character that just came in is less than 47 (I think), less than 48, it's not a number. If the code is bigger than 57, it's not a number." And the ASCII code for an A is 64 or 65, it's not a good boundary.

If I wanted to see whether IsDigit was making a comparison in the wrong place to accept or reject numbers, I'd look for a character whose ASCII code was 58 or whose ASCII code was 47. I think it's a semicolon and a slash. Those aren't letters, they're not intuitively obvious but they're the ones that if you put numbers on a linear dimension where we say this is the encoding for things that come off keyboards.

Then the things that are most similar to numbers turn out to be semicolon and slash and those would the ones I test.

I wouldn’t expect to see those to come up in a test ideas list. But if I was the person working from this test ideas list, then I would ask that on each one of these is there something more powerful that I could do with a real number than the one that is listed here? 20.0 is pretty good, that might be as refined as I get. The huge number, max+1, I might stop at that or I might say, gee, if it will accept something that is three digits, maybe I’ll try 100. If it will accept three digits and I want to go to something that is huge, suppose our max is 65535, I might try 65535, and then 65536, and then I might go to Notepad and create something that is 65535 characters/digits, copy it, and paste that into the input buffer and see what happens.

Stunningly, if you can put three characters in and have them echoed on the screen and then rejected you can often paste a remarkable number of digits in and get yourself a buffer overflow. Buffer overflows, of course, are the most common security violations that we have or the root for the most common break-ins that we have.

So huge numbers, there are a lot of huge numbers that can be interesting procedures that the person who is going from this can say, “What do I know about huge numbers?” Leading space, space point is pretty good, I might start with space two to see how it handles spaces. And if it let me get away with space 20, then it’s time for space space 20, and space space space 20, and 65535 spaces 20, we’ll paste that in and see what happens, see if it chokes.

So I can come up with a lot of nastier test cases but I’m guided as an editor as the person who’s coming up with what is a little more powerful version than this, I’m guided by these categories. The brainstorming gives me the categories. That’s the creativity that’s hard to come up with. What’s a risk? Oh, gee, that’s a risk – I can come with a nasty test case to face that one. But the group of possible risks is the group that the first brainstorm helps us out with a lot.

Here are some of the common answers people come up for this kind of an exercise. And notice if I was creating a format that I was going to keep, notice the brainstorming format, the format that’s going to come out of the list that the company publishes and keeps, I’d want three columns. Give me a specific example, tell me what’s useful about that, and tell me what I expect.

And this is going to generalize, for example, the 20 that we saw wasn’t 20, the 20 was the lower bound. If we had 10, we would have said 10. If we went through the entire analysis and said what about 10-60 then we would have ended up with test is 10. Why is it interesting? It’s the lower bound. So instead of saying that, we should just say LB, lower bound. UB, upper bound. The format is the same. But if we want to have the general test ideas list that is reusable across products, we have to substitute some variables. And here’s a list that is going into the next iteration of RUP that handles numeric input fields. This is a fairly standard version of the test ideas list for numeric input fields. Let’s try “completely nothing”.

BTW you might not be able to do completely nothing off the start because there might be a default character in the field so we might end up at “clear the default value and empty the field” then try nothing. Try any valid value, try something at the lower bound, try something UB, upper bound and so forth. So here we have is a list of too many things to do for a numeric input field, but all of them good ideas. And we’re going to come back in a second, I’m going to show these in a matrix form, we’re going to see how to delegate this, and I’m going to suggest ways to minimize or limit the number of cases you actually run. Before we do that, let’s think a little further on the general question of where do these test ideas come from.

I saw you have a puzzled look. {I have this enormous list. If I run all of them I run out of time. If I don’t run all of them then one that fails the field is the one that they’ll ding you about. And so, I’m in a dilemma here. I’d rather not generate these lists and remain in ignorance than generate these lists and say, okay just skip these.}

Slide 2.21: Discussion 2.4: Where Would You Use Test Ideas Lists?

When we think of dimensions of quality, we have to question, how do we test for each of those dimensions and we have to come up with a set of notions for each dimension of tests that might be effective. Those notions boil down to collections of test ideas.

In RUP, the Test Ideas List is input to the activities:

- •Implement Test
- •Determine Test Results
- •Define Test Details
- •Develop Test Guidelines
- •Identify Test Ideas

What are other sources of test ideas lists?

- •Bug lists (for example, Testing Computer Software's appendix, and BugNet)
- •Business domain. For example, walk through the auction web site, the shopping cart, customer service app, and for each one, list a series of related ideas for testing.
- •Technology framework: COM, J2EE, …
- •Fault models
- •Representative exemplars (such as the "best" examples of devices to use for compatibility and configuration testing. *Testing Computer Software* illustrates this in its chapter on printer testing.)

Slide 2.23: Identify a Generic List of Test Ideas

The question I have from here is, “How do we generate questions from test ideas. Now I could come up with a brainstorming process to have us try to figure out what a good population of accessibility tests would be. I can tell you from experience that would be a terrible place to start not just in a course but also in consulting to an organization that wanted to do test idea collections. You need to develop your skill, there’s a lot of skill in test idea lists, you need to develop your skill on the simple things. Once a group can make a good set of simple test ideas, they can come up with more complex questions and work on those. But I’ve talked about a brainstorming process that you might want to use in your company to develop this stuff.

Slide 2.24: A Catalog of Test Ideas for Integer-Input tests

These lists turn out to be proprietary. Companies that develop these invest a lot of time and money in them and don't publish them. So many companies put together test idea lists for how to test printer compatibility, video card compatibility, network configurations, file handling problems of various kinds. There were a slew of different test idea lists unpublished for data fields and the ones I saw were general better than the test idea lists I saw that were published. Every company is going to have some very complex things that they would like to capture a list of test ideas for but if you go to the nasty ones first, you won't get it done.

You're better off starting off with something simple. Getting good at this, by having sample test idea lists for simple things, against reliability it's good to have the simple things covered. But once get practice with this you can go to more complex issues and come up with a similar quality lists.

Slide 2.25: The Test Ideas Catalog

Numeric input fields are not the only input fields or aren't the only things you can do. One of the intriguing test ideas list that I worked on asked the question, What if we're trying to save to a file and it fails. What does the error handling look like?

Now here's some of the test ideas that came into that list: The janitor is doing the vacuuming and knocks out the internet connection in the middle of trying to save, trying to save and it failed. A remote hard disk runs out of storage versus a local hard disk. The local hard disk might be a little easier to check if you have space or not. The power goes out in the middle of saving. The disk is full or almost full but the attempt to save was an attempt that involved a file automatically saved by the program, something where it saved internal variables instead of a save where you explicitly said save this. You're still not going to make it onto the disk. There's a different risk there because the program is trying to do its housekeeping instead of you trying to do yours.

So, we came up with that by drawing pictures of all the places disks could live, all the ways we might save information to disk, and all the ways that a disk could try to accept data and discover that it couldn't do it. And then we started drawing "X" through every one of these and said, okay, here's a failure point, here's a failure point, here's a failure point, not enough room, not enough power, not enough connection, and so forth. That became a pretty efficient generator. But that was a model of the system that we were able to draw.

That became a pretty efficient way to describe a system for us and from there the test ideas pretty well fell out. You might come up with test ideas lists by reading specs, you might come up with test ideas lists by going through if you've got a mature product by going through the tech support database and ask the question, what are people calling about.

Slide 2.26: Apply a Test Ideas Catalog Using a Test Matrix

I like to publish the list in a matrix form. So across the top I have the things I'm testing for, or could be testing for, and then I'm going to list all the fields I could test in. In practice, I might have one page that lists all the relevant fields for one dialogue and then a different page for the next dialogue and so forth.

In practice, this is a physical piece of paper. It photocopies, you have a stack of them. And one of the best uses of these is you bring someone in who is a moderately experienced tester into the project and they don't know much about the product at this point and you say, okay, go figure out how to test these kinds of fields. If they're moderately experienced, they know how to work with a chart like this and it takes you 10 minutes and they go, oh, okay.

You can add people like that to the end of the project safely if you have something good for them to do. You cannot add people like that to the end of the project safely if you don't have organized stuff for them to do because you'll spend so much time training them that you will never get anything done.

Now, whether it's somebody new who comes in or you are early in the test and you are guiding yourself, my approach from here is I actually use pink markers and green markers/highlighters go through and test a field and mark it in pink if I ran the test and it failed, mark it in green if I ran the test and it passed and skip it if I didn't run the test. And what you're going to see at the end of running this is that every column has some tests in it. I tested some fields for upper bound +1.

Every row has some tests in it. But no row has all the tests and no column has all the fields because we don't have time. *What I've done is sampling on both dimensions.*

If someone comes back to me and says, you only sampled you didn't exhaustively test, I'm going to ding you. Then I show them a chart like that and say, look, I treated each of these like a risk area and I did a sample and there were no problems. Do you really want me to spend at least 20 times as long, 20 times my budget so that I can go through every possible test and run them all?

At that point, a reasonable executive is going to say, Oh, I see what you were doing. That's very efficient. Congratulations, I understand we can't do everything. And an unreasonable executive is just going to be unreasonable. But this is an organized and easily communicated process and in my experience people look at it and say, yeah, I don't think I could have done anything more efficient than that and more reasonable than that.

Every time somebody calls about a problem that is an actual error in the product, you have a class of test cases that you seem not to have run. Maybe that's not true and you ran the class, but you sampled and just missed this one by bad luck. But more often, you just missed the idea for that test so now you have a generator for a new set of ideas. So catalog or lists of test ideas could be generated on your own from a model or they could be generated in a group or both, have the group work up the model and then work from that. The value of the catalog is its reusability across many, many projects. That's what makes it worth spending your time getting something that has a lot of stuff in it and formatting it so that it can be easily reused.

Slide 2.27: Exercise 2.5: Your Own Test Ideas Lists

Pick a topic of interest to the room and expand into test ideas list and matrix of ideas. Doesn't need to be input fields (probably shouldn't be.)

If the students don't have ideas, suggest installation testing as a technical example; internet stock trading as a business domain example.

NOTE: This exercise is most successful if you suggest a topic one day, encourage people to spend 15 minutes generating test ideas on their own overnight, and do the actual brainstorming session the next morning.

Does context of the application help the brainstorming?

If I'm trying to build a test ideas list that could be reused across many applications, then in the ideal case the more context free I can make my list the happier I'll be because it will be more easily reusable across a broader context of applications. In practice though, the brainstorming session to come up with these have only worked, I've facilitated a variety of groups trying to develop lists like these. Independent test labs, how does an independent test lab work?

They go to a company and say, we're an independent test lab. That's all we do for a living – that makes us experts. We have lots and lots of little experts we can rent out to you because we're an outsource agency and that means we're cheaper than having employees in-house. That's the pitch, right?

How can they be so cheaper and manage their overhead? The answer is they hire people for $10 bucks an hour and they charge you $30 - $40 bucks an hour. For those of you who think that's a big markup, I should point out to you that every independent test lab I've ever worked with who only charged a multiple of 3, 10 to 30, has gone out of business. It's not enough. The amount of training support, executive support, facilities, and so forth, eats that money like you wouldn't believe.

So here's an organization that can't really afford to discount much who's going to sell you a tester for $50 bucks an hour, okay, so they can pay $16 bucks. I don't know what you can find for $16 bucks an hour but in California where I've been seeing most of this done, $16 bucks an hour will not hire you people with 20 years testing experience who are good. So how do you make people who came in last week to your lab and agreed to work for $10-15 bucks an hour look really productive on the first day.

And a lot of the answer comes with test ideas catalogs. You take these new folks who have some background in software, if you're dealing with total juniors, then you have a supervision cost and maybe you're going to pay that supervision cost, but if you're dealing with mid-level testers then you take some test ideas, you sit them on this product.

The product comes in on Monday, you say, uh oh, let's find some bugs. You take a bunch of very standard tests, you throw people at those tests. You generate a ton of bug reports, through tons of bugs reports back at the development group, and they say WOW. You catch your breath and try to figure out how to test this thing. So the broader range of test ideas you have canned, the more you can sell your services as if you were really experts. And so test ideas list are in fact, are the valuable intellectual capital of some of the best test labs in the country.

Now, if you're dealing with a test ideas list generation process like brainstorming, with people who have never attacked that type of problem before, you are going to spend hours, and hours, and hours, in brainstorming coming up with a second rate list. Figuring out what's really relevant and what's not relevant isn't going to work. On the other hand, if people have a lot of experience with this and they go, oh, yeah, write to a bad file problem or write to a file and don't succeed in saving it properly.

Even though that's kind of an esoteric area, if you've done that kind of testing 28 times, and you go, *that's such a boring thing, I've done it 28 times boom, it's a wonderful opportunity for a test ideas list.* If you are dealing with folks who don't have that range of experience in something you want to capture. You have to go through some specific contexts to bound what they're going to come up with. Just like we didn't start with let's come up with a test ideas list for a numeric input field where it's lower bound and upper bound, we could have. We came in with let's think of a database application with 20-50.

You might end up with a brainstorm of essentially the same problem. But here's this particular problem for this product. Now let's look at this same problem for this other product. Now look at the same problem for this other product. What did we just extract? The context is either in the head of the people who are already there, many contexts so you can come up with general questions, or else you have to go find context as part of your building process or building the final list.

Multiple contexts are essential. But if you're lucky, the people you are working with will already have those multiple contexts in their heads and so you don't have to confront context in your list. The list that gets output needs to be as context-free as it can be. But the inputs are not well educated unless people have done them in real systems, seen what works, and from a variety of contexts can tell us which were the most valuable things.

Module 4: Define Evaluation Mission

In the Module 2, we talked general about the notions of quality, the notions of testing, the notions of quality categories, and then we burrowed down to test design and the bottom up view of test design lets us come up with test ideas. Those concepts give us a nice background to look at the workflow in RUP. All of the modules after this go through the workflow of the RUP. In Module 3, we look at a broad overview of the testing related workflow of RUP and how general testing is handled in RUP.

This will be a quick familiarization.

Slide 4.2: Module 4 Content Outline

The first workflow for testing in RUP is the one that defines the evaluation measures.

So what we're going to do in this outline is to go very quickly through the definition of the workflow. Notice the definitions of the activities and the artifacts. When I say notice, I mean those are in your text notes. I'm not going to through all those in detail.

Slide 4.3-5: Workflow: Define Evaluation Mission

In general, the structure of these modules is a RUP transition in each module. Basically what we have is a quick peek at RUP followed by statement – let's focus on one or two things in this workflow and I say – let's focus on…, then I start focusing. The place where people should have understood that structure was either in in Module 3.

The instructor is going to have to make it clear that there is a pattern we see with every workflow. The pattern goes like this. The workflow talks about all sorts of activities done by all sorts of people. We can't begin to teach all those things in a 3 three-day class. What we can do is take the one or two things within this workflow that we think face-to-face instruction might really help you understand what's special about that workflow and work on those.

What I'm going to do is focus on the specific most important question in this workflow: How do we figure out what the mission for test group should be? The rest of the stuff people can get as guidance from RUP and there are ideas in the course notes for them. And certainly there are lots more in *Lessons Learned in Software Testing*.

So Workflow Define Evaluation Mission is defined in terms of identifying the appropriate focus for the testing effort and gaining agreement with stakeholders. The other key piece to recognize is as well every else in RUP, this is an iterative process.

You want to check every iteration – is the mission the right mission. We will see as we go through, that the mission for the test group in fact changes throughout the release and it should. Yes.

Even if you say, "We don't develop software iteratively so that won't work for me," this ocnsideration still applies. The short answer I'm going to give you is that test groups end up facing changes in their mission over the life cycle of the product whether that's explicit or not. We will see that in an example, but I'm going to defer that and come back to until after we see the example. I hope you will say, oh yeah, I see that. If not, then we should talk about it then.

Slide 4.7: Exercise 4.1: Which Group is Better?

Let's hit the mission question. I want to hit this with a simple example, an artificial example, I want you to have some patience with the artificial nature of it. To get to the question of mission in the way I want to get to it, we've had to simplify everything else enough that we don't get bogged down in all sorts of irrelevant facts.

Imagine that we had two different test groups that were given exactly the same program and imagine that we went through this program and found five areas that were interesting and had some risk. There are a bunch of other areas of the program that were less interesting. But there are five we look at and say these were equally important, we care about all of them, and additionally we have some reason to believe, that on average a defect in any one of these areas would be as severe as a defect in any other of one of the areas.

Here's what happens with these groups. We give the program to Group 1 and Group 1 follows fairly normal testing strategy. They start out looking at all five functions lightly, check to see the functions are there, test the functions with easy values and basic happy paths. Functions B, C, D, E pass. That doesn't mean they pass everything you could ever through at them, on a simple look there's nothing obviously wrong.

Function A on the other hand starts out sick and stays sick. All through testing, Function A is the bad function. In fact it's such a bad function, that the test group even thought they try to allocate time to B, C, D, E and they do a little more testing on B, C, D, E. They haven't ignored B, C, D, E but this is a group that's motivated toward finding all the bugs they can and every time they turn around, more nests are being found in Function A. Every time they send bug reports back to the programmers and get reports back, this is supposedly fixed, they are finding side effects coming out of Function A. and so most of their attention gets spent on Function A

At the end of testing, Group 1 found 100 bugs in A and none in B, C, D, E.

Group 2 has a different strategy. They start with the same strategy. Most groups start with the same strategy; let's wander through all the functions and do some light testing quickly to find the areas that might be at the greatest risk. So they see that Function A is pretty bad too and they allocate a little more time to Function A. But overall their testing philosophy is we have a minimum level of testing for anything we care about, a minimum level of coverage that we're going to hit for anything we care about and it's a substantial amount and so they work pretty hard on B, C, D, E and they find six bugs apiece. There were bugs there, it just took them more work to find them.

Of course, if you spend your time on B, C, D, E there less time to spend on Function A so they only found 50 bugs on Function A. The first group found 100 bugs, the second group found 74. Which group did the better job of testing? You say Group 2 – Why?

Let me clarify the facts. Group 2 kept testing Function A they just didn't spend as much time on it. If you had 5 people to test, Group 1 spent almost all those 5 people's time on A; Group 2 kept cranking on A but they spent more time on the others. That meant that 50 bugs were left unfound in function A and unfixed. Anybody have any thoughts, let me take a show of hands. How many folks think Group 2 was the better group? About half.

How many think Group 1 was better? 3 – Why do you think Group 1 was the better group? *They found more bugs and all bugs are equally significant.*

Of all you folks who thought Group 2 was the better group, what do you think of that? Who's a Group 2 person here? *Coverage was better in Group 2 and that's better results because the functions are equally important.*

The 3 of you who thought Group 1 was the better group, what do you think of that? Your hand was up, what do you think? *End users will find fewer bugs.*

So remember that for both groups everybody tested the happy paths for all the functions. We're not talking about groups that are fundamentally incompetent. They started out saying what's the purpose of this function, let me check the basic purpose with reasonable values and Function A failed for both, B, C, D, E passed for both on those tests.

So now they're getting into harsher testing. Group 1 allocated their time here and Group 2 allocated their time into harsher testing into B, C, D, E and less harsh testing in A. For both groups, it is unlikely that just walking in and pressing a key will stop you in the water. They did test it they just didn't do as thorough a job in one set as the other.

This is all the testing for the project. In fact, let's go to the next slide. Here we are six months later.

Slide 4.9: Exercise 4.2: Which Group is Better?

So Group 1 shipped the product, Group 2 shipped the product. You know, some companies operate on the schedule that says on February 1 we will ship the product and on February 1 Group 1 had found 100 bugs in A and nothing else. These guys found 100 but missed all 48 that were found outside of Function A. This is the product that got shipped.

On February 1 Group 2 found 74 bugs in A, B, C, D, E. Now what's really in the product was 148 bugs. These guys spent their time and found half the bugs in A and half the bugs in B, C, D, E. So 74 bugs got found and 74 bugs got missed. 74 bugs got missed out of a total of 148.

So let me ask again, how many think Group 1 did the better job? A few more. How many think Group 2 did the better job? Several. Why do you think Group 2 did the better job? Group 1 found more bugs faster and so that of course is a good thing, but Group 2 covered B, C, D, E and assured a broader range of use flows are likely to work. Any group 1 advocates?

So Group 1 probably has better negotiating power to get Function A fixed than Group 2. What if it turned out, what if you were in a company where every significant bug that got found, a serious attempt was made to fix it. The company had a corporate standard which was, we don't leave bugs that we know about unfixed unless desperate circumstances happen.

So in this case, the 100 bugs were fixed and in this case the 74 were fixed. Would this change your point of view? It would make you sit on the fence on the issue.

The statement is that Group 1 found 100 leaving 48 in the field. I'm going to add one tiny note to that, you mention the notion of metrics and I'll mention the notion of customer call costs, if we assume the bugs were equally significant, then the tech support budget associated with Group 1 was probably lower than the tech support budget associated with Group 2. And so by that measure, Group 1 did more successful testing than Group 2. And yet, we still have folks that want to advocate for the better effectiveness of Group 2.

So how do you argue against that? Group 2 found half versus either 100% or 0.

But I want to address a different point which was that at some point when you have a function that is in bad shape, the wisest thing for a tester to do is say, we're not going to test this anymore, this is just not stable enough to be worth testing. That's true. And another factor no one has mentioned but is certainly true, when you find one bug, the fix to that fixes something that would have been found and reported if you'd continued testing. On the other hand, I certainly have had the experience of telling people that I'm done testing an area, that it is broken and they need to fix it, watched them not to do any more work on it because there was nothing else listed to do on their todo list and watched the product go out with weaknesses in the area where I'd documented fundamental weaknesses.

It depends on the company. If you think that merely saying this is so badly broken I'm going to stop testing is good enough, and in some companies it is, in some companies after finding 10 bugs in 10 minutes in Function A, you could toss it back and say, give me a complete rewrite, we're done. And the programming team would say, yup, you're right, we need to start over. Yet, in other companies that's just not going to happen.

The example is artificial to keep all the numbers around it easily calculated. The experience is that some test groups have the vision that they're supposed to find the most bugs and the heuristics that where there have been bugs before, there are more bugs. That's a very common heuristic. A heuristic is a rule of thumb. And one of the first things that has been taught to testers in all of the testing textbooks point out that all of the areas that had bugs before are probably still unstable are and worth devoting more test time to. If you follow that approach and take your time away from investigating other areas.

It's zero-sums game as far as time is concerned -- if you spend more on one thing you spend less on something else. Everything we spend chasing bugs in one area, we take away from broad coverage of the others. All I've done is taken that to the logical extreme so that we can see that as a real contrasting strategy.

Let me hit the contrasting strategy a little different way. I want you to pretend for the moment that you are the Project Manager. I'm going to come to you as the Test Manager for Group 1. BTW, the ship date is February 1, it is now January 23, basically we're done. Here's my quality status report: *Function A was really sick. You should never have given me this code and having given me this code, you should have taken it back a long time ago but you didn't. We've finally beaten it into submission. As far as we can tell, Function A looks okay now. We've lightly tested Functions B, C, D, E. The main use cases work and as to the rest, we've sampled but we don't have a lot of data. That's Group 1. Now think of yourself as Project Manager and it's January 23. I am the Test*

Manager for Group 2. Let me tell you the status of this program. The mean time between bugs discovered in Function A is still about one per hour. By the time we finish this meeting we'll have another one. We don't know when it will end but it probably won't be tomorrow. Functions B, C, D, E had some bugs, we found some bugs, there are probably some more bugs, they're getting hard to find.

As the Project Manager, which do you like better, Group 1 or Group 2? Why do you like Group 1 better? Your focused your time on the Function that worked least well, drove your test case through your testing for use cases for all functions and they appear to be more or less okay. What else would you expect from the test group?

You're the Project Manager, do you like Group 2 better? Who is in that category? Group 2 covered more of the alternate paths and enabled a stronger statement about quality than Group 1. Not stronger bad or good, but stronger in terms of more informative. And a strategy that can enable a stronger statement about quality is a better strategy than a strategy that merely finds more bugs. Is that a fair summary?

Group 1 is in a far better position to ship because they're saying they're not in a crisis, of course, Group 1 hasn't discovered the bugs in B, C, D, E that might make it nervous.

Whereas Group 2 can say to the Project Manager, you're in trouble. Now let's add yet another twist to this. Suppose that you were a PM that was indeed a hard, fixed ship date. It will not ship after February 1. Whatever you find is what you find. Whatever you find is what we fix. We will fix nothing you don't find. And we find nothing you don't find and we will ship February 1 even if it is erasing people's hard disks. Under those circumstances, which do you care about? Maximum number of bugs found or best quality report? How many folks think that under that circumstance you are looking for max bugs? A few. And how many of you think that you're looking for a quality report under that circumstance? One.

The circumstance is an absolute fixed ship date, the only bugs that will get fixed are those that get reported, so the test group that finds more bugs, get more bugs fixed. But if the test group only found 2, the product would ship. If the test group found 400, the product would ship. The ship date is totally independent from what the test group would find. Do you need a quality assessment from the test group at that point or do you need bug fixes?

How many folks think they need bug fixes more than they need a quality assessment? Some of the folks that liked Group 2 say, oh in that case, you need bug fixes more than quality assessment. How many of you think we still need quality assessment more than bug fixes. 3-4 of you think we need quality assessments than bug reports.

Okay. I want to suggest to the four of you who said that, ***you have a conflict of mission with the Project Manager.*** Because the Project Manager probably couldn't care less about your quality assessment. The Project Manager is definitely trying to get bugs found and very minute spent assessing quality, the Project Manager is going to sit back and say, *What are you doing? Spending time when you could be finding all these bugs and you're not! I don't care about your report – just find the bugs*.

Now on the other side, I can say from my experience that one of the less pleasant moments that I had as a Project Manager was getting a Group 1 type of report and writing a note back that I was going to slip the schedule for two weeks to allow the test group could do its job which was to find enough information about the product so I could decide whether to ship or not. The distinction was that I was not facing a hard ship date, I was facing an embarrassing schedule slip, but I had the authority to slip the schedule. At that point the testing group was a technical information provider of the kind that could help me make the decision about ship instead of the only the kind that could help me make the decision about what to fix.

For a Project Manager who has flexibility based on a quality assessment and wants that quality assessment from the test group, Group 1 is off mission. Group 2 is on. ***The worst mistake you can make as a Test Manager is to cross missions.***

Group 1 and Group 2 are both fine groups *for different Project Managers.* If you have a Project Manager that has no value for your quality assessment, and Project Managers who in good faith are up against certain kinds of deadlines simply do not have a value for your quality assessment. For that person, the more time you spend on assessment and the less time you spend finding bugs your not serving them and they're going to wonder what you are doing. On the other hand, for the person who is looking to you for an overall assessment the more time you spend just finding bugs and the less time you spend finding out what the overall product looks like the less helpful you are being.

Now I mentioned the notion that people's mission changes over time. Imagine there's still a February 1 ship date but imagine we had these data in December. In December both groups might be hunting for bugs and both groups might be taking different strategies for finding out where the real nests are. Group 2 might be saying you know a little broader coverage might give us better risk management overall and Group 1 says let's clean up one, then the next worst one, then the next worst one.

Both of them early on probably have the main mission: find bugs, get them fixed. And as the ship date approaches the group that might be looked at for assessment is going to be under increasing pressure to stop looking merely for bugs and start looking for indicators of quality. An indicator of quality might be, I tested as thoroughly as I could for three days in this area and nothing was found. We can't find problems in this area. That's an indicator of quality for that area. If I have a hunch that there's nothing there and I spend ½ day and there's nothing there, my task would be find the most bugs I could, that would be the end of that area.

But if I was trying to do an assessment, I might do the whole three days and come back with a more thorough assessment. I will spend more time perhaps looking at the lower pay off areas if I was trying to gauge how reliable the product is in those areas than I'll spend if I'm just trying to find the maximum number of bugs.

As my mission shifts from find the largest number of problems because the programmers are ready to get into help me understand what this product really is and what my schedule risks are, I'm going to change how I'm spending my time from being a Group 1 to being a Group 2.

And that's true even in a company that practices waterfall development, a company that follows zero iteration development, the test group is still going to get builds. And every time a new build comes in it's worth the tester saying, what should we focus on, why should we focus on it, how would this serve the needs of the Project Manager and the rest of the project team this week. And the answer to that question might be a different answer each week of the process even though from a Project Manager's point of view it's one unified process.

One of the advantages of Group 2 is lack of surprise. If you really had a test group that operated in series, will thoroughly test the highest risk area, the test the next highest risk area, and so forth and the Project Manager doesn't understand that, this coordination of mission is very important. If the Project Management doesn't understand that's the approach of the test group, then you'll have a situation where at the end of December, Function A had a 100 bugs reported and B, C, D, E had nothing and then two weeks into January, the Project Manager is seeing bugs in B and saying I knew there were no bugs in B.

On the other hand, if you have a common agreement about a mission, the PM says I only have two programmers and I want to focus their time everything on Function A, everything on Function B, I don't want to have people shifting their attention back and forth. Get to an area, beat it up as hard as you can. Go on to the next area, that's going to help me keep my staff focused on related problems. Then the Group 1 approach in series would be exactly what's needed. It depends on what the two groups have.

We provide information to people who use the information and we need to provide the information in the way that they can best use it. There is no test group mission independent of the needs of the organization for the services we provide. We're a service organization.

The Project Manager might have a heart attack at finding out later on that there are problems in these functions. The statement is the Project Manager's product still has the better methods. Now in this example that happens to turn out to be true. The problem for me as the Project Manager or you as the Project Manager is that as you look at these numbers, you don't yet know that these will be the numbers and when they become the numbers, the question you get to ask yourself is: "Was this by skill or by luck?"

In the case of Group 2, there's a lot less room for good or bad luck. Group 2 is sacrificing the total number of bugs found in return for less surprise. In a group that values a reduction of surprise Group 2 is better. In a group that values minimization of total bugs but is tolerant of surprise, Group 1 is better. Management is never tolerant of surprise, some management is willing that they face risks and that it is necessarily the case when you take a risk that some of the risk might bite you. And so they may have a tolerance for a certain number of them.

So let me just speak again as a Project Manager that has made this decision and doesn't regret the decision. At the end of the day for me, when I faced the Group 1, I stopped ship on the product not because there were bugs but because there was too much uncertainty for me to be willing to ship and while what they found probably would not have justified me stopping the ship, that is I wasted two weeks. I was not willing to sign my name on the release until I had more information about the areas I considered under tested. At the end of the day, it didn't matter to me what the metrics were, it mattered to me what information I didn't have. As the person in charge of making the release decision, I was counting on the test group to give me the information they didn't provide.

Now was I managing properly? The way that I was managing improperly, was that somehow I had not succeeded in working with the test organization to make sure we both had the same mission. I wasn't able to get them to not do their thing in their way until finally I said, I have the ship/don't ship decision and we don't ship. Now I have been in the exact other role as Project Manager where all I wanted was bugs and I got frustrated with the process. It is not a question of what is the one true way, the question is how do we achieve congruence in the organization. And the critical thing a Test Manager has to consider, you might decide that in your sense of your integrity, and your sense of your group's integrity, and your sense of your responsibility to the people other than the Project Manager to a broader corporate mission, there's some guns to be stuck to and you're going to go and do it this way even though you have a Project Manager who's asking you to veer off in another direction.

So when you make that decision what I want to urge you to do is make that decision absolutely consciously. This is the mission that would best serve the Project Manager and this is the best mission that I believe best serves the customer or the company, and this is the one that I'm going to choose or this is the one that I'm going to choose. But understand where the pull is and how comfortable you are with it. That's the hard part that we need to do to define what our tasks are going to be for the rest of the testing project.

The comment was, in Group 2 they made their ship date and were willing to absorb the cost of more bugs in the field, that might be one success. Here's a different success. Imagine as the Project Manager I go the Vice President of Development and say, let me explain why I need to slip the ship date for two weeks. Now when I slipped for two weeks, the way I had to slip was say the test group did not do its job, we had zero information about this, those people are causing a slip based on their incompetence. Let me just give you a piece of career advice, that's not the memo to have written about your group.

If you're running Group 2 on the other hand, then you have some cover. I get to walk up to the Vice President of Development and say, you know what? Every hour we find a new crash in this. It's a problem. It takes us longer to find a problem here but we know there are a few left. I think we should hold it for another week and see if it gets better because it's just not commercial yet. That's very different from I can't tell that I don't trust it and they haven't given me any basis for confidence.

Here we have an assessment and say, this is probably what it will cost us. Tech Support can sit with me and say, this is $100K/year, this is $1M/year, this is whatever it is per year in probable problems and we've can make a cost/benefit decision. With Group 1, on the other hand, we start consulting astrological tables to make our ship decision. If we have flexibility in the ship date, this is fundamentally uncomfortable. And Group 2 gives me the basis to make a reasonable decision. Group 1 doesn't.

Of course these numbers are oversimplified. But I want to suggest to you the strategy behind the numbers is an appropriate strategy. And that works like this, if you know that you are facing a firm ship date with zero flexibility and you also know your Project Manager is not much interested in an assessment, BTW those are separate things, you might for example on facing your firm ship date still be able to predict tech support costs and the value of being able to staff tech support appropriately and train tech support appropriately might be so high in your company that they're happy to have you do quote "less efficient", in terms of bug metrics, testing in order to be able to make quality predictions.

Some companies want that, some companies don't care. A few companies I know in the last two weeks before release, assign two or three tech support folks to do their own assessment and to do their own estimation. They don't trust the test group's numbers, they don't care about the test group's numbers, they come up with their own.

So in that company, whether your like that style or not, that company is not being blind, they're just managing risk in a certain way without using the test group. In that company, you don't have influence over the ship date, you are not relied on to give an assessment of product, what they want you to do is hunt bugs. Your task is to hunt in the fastest way you can for the worst bugs you can find and of course to check the areas you think are unlikely to fail lightly, briefly to make sure that there are no easy pickings.

Certainly, if I was testing Function A I wouldn't spend all of my time on Function A for every build. I'd do a little into Function A, I'd get enough to say you have rot, I'd leave it. I'd go into the next thing and see what they've got done. As soon as they come back to me and say they've got A more or less okay now, I'd go back and discover you still have rot then pull off and go on to the other stuff. You can manage this with some efficiency and still say that every time it is sensible to test Function A, I will. That's my number 1 priority area. Every time it is sensible for me to test the next highest risk area, I will. That's my number 2 priority area. And just keep hammering back on those areas based on what you think you are likely to find. And while I agree that you will be unproductive to go past a certain level in any area because you are likely to generate enough change within the code that anything you find after this point won't map onto the code when it's redone. You'll still focus your time on where's the next serious bug likely to be. Fundamentally a Group 1 strategy, and fundamentally not a Group 2 strategy.

The Group 2 strategy is: Where is the next most valuable piece of information about the status of the project likely to be. That might be in finding a bug and it might be in not finding a bug. Group 1 not finding a bug has no value but in Group 2 not finding a bug carries something.

Every build you have to reassess your test mission. Every build it's worth sitting down with the other stakeholders and asking, are you getting value from what we're giving you? If you have a Project Manager who feels the most important thing you could have given him in the last week were all those integer tests, then collaboration has had the value that that Manager was hoping for but probably this week they will ask for something else.

Every build it's worth asking should we be doing this? And you know early in testing, very early in testing, you have the very strong question, function by function is this thing working okay. We would like to be in a company where they do enough of their own unit testing on the programming side that most of the really simple tests just pass. In a lot of companies, that's true. The obvious problems, the simple boundary tests and so forth rarely generate bugs. And in some companies that's not true and so you end up in the situation where in a company where there's a lot of risk associated with some elementary operations, do you spend a lot of time testing them, well maybe you do and maybe you're not dealing with code that this current group has a lot of control over.

During the Y2K projects, people found some wild problems. They would change some code that was obviously, completely and totally unrelated to everything else and break everything else in horrible ways and not understand how to fix it because all sorts of the changes had been made years ago by people no longer there by patches. And so the source code everybody had was a more or less approximation to what was running inside a computer that was the company's financial record keeping system, lightly documented. In that world, people ended up having to do all sorts of crazy regression testing at very basic levels. There was no such thing as unit testing, nobody had source let alone the ability to test at the source level.

So, there are circumstances where running the hundred test ideas for the integer field is not irrational. If you're in those circumstances and you are not doing that, it's a problem.

It's the congruence of mission issue. And if you're not congruent you fail. If you are congruent, you may need a lot more time, jointly, than you budgeted so far. But if you are congruent and you're spending a lot of time on this stuff let alone the complex stuff, at some point the Project Manager is going to come back, not just the Test Manager, and say, we need more time, more money, and more people.

Whereas if you are non-congruent you are finding problems where the Project Manager says I wish you'd spend more time over here. And you say, can I have more testing staff? They're likely to say, no I'd rather give you fewer and have you work on the mission I want for you. And as the Project Manager, the person responsible for getting the product shipped on time at the right quality level, within budget, with the right features, it might even be appropriate for the Project Manager to say, you are overspending on the wrong services on my project. I'd rather give you less.

It depends on the company and how they view the management structure for a project. But somebody has that ultimate responsibility. And that somebody should have the right to say that an organization is not operating congruently with the management philosophy with that project maybe doesn't get to spend much more even if they're finding problems they think are important. I need to close this discussion down. I think the point has gotten across about mission.

Slide 4.10: So? Purpose of Testing?

We’ve made a contrast between the two most common missions of testing groups: find bugs and assess quality. I can’t count the number of times I’ve seen debates between:

- “Our job is to find bugs. A test that didn’t find a bug is a waste of time. Only bug finding has value.”
- “Our job is quality control. A process that doesn’t lead us to an estimate of reliability is a broken process.”

And it get s vituperative: “These guys are negative!” “And these guys are process weenies!”

The real issue is congruence, not right or wrong. But at the point where we think about congruence in general, we recognize that in fact there are really many different missions in the field of test organizations. The same test organization might go through many of these missions through the course of the project, but some companies will focus their test group primarily on one of these missions throughout the project.

Slide 4.11: Varying Missions of Test Groups

Let's look at a few of these. Find defects we know. Maximize the bug count – here's the group that wants to prove it is the primo, suprimo best metrics group in the organization – we can find more bugs than anybody. BTW, in shootouts between independent test labs -- I'll give four labs the code and try to figure out which of the four labs I want to retain -- maximize bug count might be exactly the mission.

Block premature product release – your task if you have a company where executives have announced that on January 1 the product will ship. There's no way the product, well, it could ship January 1 if the company doesn't mind getting sued, but there's no way that a reasonable person would ship it on January 1. Sometimes the test group gets to play the role of organization standing on the railroad tracks advising the train to slow down or stop. Now most of the time that's an interesting way to get flattened out. You can do exercises or you can stand in front of trains, eventually things will get redistributed.

If you decide you're going to block the train, you may as well do it well. If that's what you've decided is your goal, then blocking premature product release does not necessarily mean finding the most bugs, it means finding the bugs that will stop ship on the product. And so, it might be that you find only one horrible bug per week, but you get to walk in to the project team meeting each week and go, guess what want to ship with this one, look at this one. And everybody else in the room go "oooooo". And while they're busy fixing that one, have your testers busy going "where's the next bad one". Your metrics might be terrible but your schedule might really get slipped.

If your strategy is blocking an out of control project, the net effect of that is if you block it long enough the executives of the company will come and do a thorough investigation of the project. Some groups adopt as their secret mission getting a Project Manager fired. And some Project Managers need that. And if that's your mission, and I'm not suggesting it should be, but if that's your mission, blocking that person's releases is part of the strategy to do that. If that is what you've decided to do, figure out a way to do that well and that also doesn't get you fired.

The assessment issue really is at two levels. Assess quality to the point where we can tell the DOD what the probable reliability level is, ±3%. It takes a much different investigation than assess quality to the point where the Project Manager can rely on her educated instincts to say, I think we're okay, we're going to ship it. We'll absorb the rest of the costs. I hope I'm right.

For many safety critical products, the question is not exhaustive testing of the device against all possible uses. Instead the question is whether there a safe way to use this product that is thoroughly documented and that we have absolute confidence in. And so, you end up with a series of happy paths and a small number of alternate paths that are the ones that are reasonable foreseeable misuse, if someone were trying to follow the instructions. The then device manufacturer sells the device to a hospital for example, and says you will use this thing only in these ways. That product is assured as management throughout the project have made sure that, if they do it in the way that they've been instructed or with any reasonable variation including wrong ones, nothing bad will happen. That level of testing is a very common level of testing for professional-use medical devices for example. And if you go off into extreme cases that no one trying to follow the instructions would do, you're not really spending time on it in a way that meets the corporate goal of getting a project that they expect to sufficiently safe. Safety is not merely safety of the device -- it's safety of the practices of using the device. And in some cases you can take a device and if used this way there is a high risk and have it be completely safe by making sure no one would use it that way.

So, a goal of making sure that a product is sufficiently safe for marketing might not involve all the kinds of testing you would do if you were trying to maximize overall reliability. Or even assess overall reliability.

Different missions. We have to figure out which one is yours.

Slide 4.12: Optional Exercise 4.3: What Is Your Mission?

Possible exercise/discussion:

If you do this as an exercise, split the class into project teams. Teams should include the test manager, product development manager, marketing manager, and the company's lawyer. For larger teams, add the tech support manager and the sales manager. If this is an in-house development group, consider replacing the marketing manager with the customer representative.

What would be an appropriate mission for testing a PBX? A computer game?

Have each group tackle the same type of product and negotiate their mission. At the end of the exercise, compare missions across groups and get to the reasoning underlying the differences in choices.

Then transition to next slide by saying,

Now let's look at mission a little differently. How many of the product's total bugs do you think the test team is finding?

For in-house teachings, you may want to skip some of the next slides.

Slide 4.13: A Different Take on Mission: Public vs. Private Bugs

I want to look at a different aspect of mission. Functional testers are doing black box testing. Sometimes people look at people doing functional testing and say, why bother? Wouldn't you be more efficient if you just read the code? You could find all sorts of bugs that you can only guess at if you're working from the outside. It is interesting to look at some statistics on bugs that are found by the programmers and bugs that are found later on by testers.

So let's think about public bugs and private bugs. A public bug is a bug that has been left in the code for anyone else in development to see. Testers can find them. If testers miss them, customers can find them. Whoever; the programmers missed them. A private bug is a bug that a programmer found herself. She made it, she found it, she fixed it, it's hers, no one notices. Counts of public bugs end up saying that average programmers make 1 bug per 100 lines or code, 3 bugs per 100 lines of code, .5 bugs per 100 lines of code – somewhere in that region. It varies a lot across individuals but that's the order of magnitude.

Private bugs rate have been reported more widely. It's pretty embarrassing to report private bug rates honestly. But one report was 1.5 bugs per 100 lines of code. I thought that particular estimate was so outrageous that I ended up working with a group of pretty skilled programmers and having us all count our bugs as we went. And after a couple of weeks we went to a Chinese restaurant and were doing a family style passing everything around, we also did family style statistics. Except we were all embarrassed to admit our actual numbers. And so instead as we went around the table, the answer was, yeah that 150 isn't so embarrassing after all. Of course, mine was a little better.

The way we counted the bugs, if you were typing away and noticed something as you were typing and backspaced over it, it didn't count. But as soon as you leaned back, lit a cigarette (this was a long time ago), started editing, as soon as you were no longer in production mode, you were once again in the look and think mode, changes you made from there went onto your tally sheet. All of those were private bugs. Most of those were found by compilers. There are lots of other tools we were able to use to find problems without testing in the traditional sense.

If you go to some data from Capers Jones, who counts or private bugs starting from the time the code is checked into the source control system, it has now been compiled successfully, he still says there are 14 bugs left per 100 lines of code. And yet, testers are only going to be finding one bug per 100 lines of code and the programmers will find the other 13.

So the folks who say, gee, if you could look at the source code you could find a lot of stuff, are kind of correct. Somebody looking at the source code could find tons, and tons, and tons, of stuff, but by the time that code gets to the tester, that road has been pretty well cleared. If you end up doing the same things the programmer did when she was testing her code, you're probably not going to find anything more than she found. And yet there are things left. The things that are left reflect the blind spots in how the programmer analyzed the program and how the programmer and her tools analyzed ways to find bugs in the program. You don't address those blind spots by using the same tools or using the same sources of information.

And so the query becomes if you take the code and give it to somebody else and say, I think I found most of it, what's left – the somebody else is going to have to attack it in a totally different way. And in the functional testing approach we say, okay, great – I'm glad that in a totally technical point of view we understand everything we can understand. But let's now look at what the stakeholders think about this product and check the point of view of every stakeholder associated with this product, does this help them meet their needs. And as we look at the code through that filter, we come up with combinations of tests that are probably not combinations the programmer thought of as she was going through the code in her way.

So yes, source code driven testing in some ways is much more powerful, but it misses stuff. On the other hand, there are some test groups that have this wild idea that if they could write test cases and automate them, then the programmers could run the test cases and then the code that comes to the test group would be better because the programmers had done as part of their build process all this testing defined by the test group.

So let's take this process that finds 99% of the problems and replace it with a process that finds 1% of the problems and say, oh this will be better. To the extent that we discourage programmers from using the techniques they are best at, instead have them use the techniques we're best at, we're risking increasing the net number of bugs that come into testing by 2 orders of magnitude. I'm not sure that's the best approach and as groups think about doing this, as groups think about giving large automated functional test suites to programmers, they might want to ask themselves, how the risk will be managed – the programmers will rely on those tests instead of relying on their own unit tests and their own other tools. You might come up with a good management strategy in your company but it is a risk that must be managed.

Am I convinced that 80% of developers do activities to clear the road in their code? Well I'm convinced that at least 100% of the programmers that use compilers discover that they have a tool that is designed to catch all sorts of errors in their code. I think that even beyond compilers, programmers use tools like Link, source code checkers. Do I think programmers will spend time doing exhaustively doing planned unit testing and specifically writing test cases before writing their code or after writing their code, I think there's a lot of variation across that. I think that many programmers want to do that and are not given enough time, many programmers want to do that and do it, the entire extreme programming community kind of does that religiously.

And then there are some programmers who have been successfully trained by their management that that is something a tester should do. But even those folks will sit back and read their listings and in many cases will pass their listing to somebody else and say, why don't you look this over. To the extent that we guide them to spend their discretionary time, or time that would have been discretionary time on tests that we would design and run anyway, whatever practices they might use the folks who are very time pressed or very uninterested in testing, whatever practices they might do, they are probably a little less likely to do. The informal code review and so forth are less likely to happen. The fact is that it is a rare programmer that is giving us more than a few bugs per 100 lines of code. And yet, even really good programmers end up if you count where they start from generating 100-200 bugs per 100 lines of code.

So they have to be doing something. My question is, at what point to we risk interfering with what they're doing instead of supplementing what they are doing and I think there is a risk to that. I think I've seen it.

Is that risk in terms of creativity or workflow? Company X had a product that the programmer staff weren't entirely enthusiastic about. The testing staff, sensing that, decided to give the programmer staff a suite of automated tests. The result of that was the programming staff didn't follow even the minimum normal practices that might follow and instead tested what they were told to test and went off and did other things. That product went off into the marketplace and was famous for being across different operating systems for that product, that port made magazines for how unreliable it was. Was it the lack of creativity or the lack of work, I think it was the sense of the programming staff that they were no longer accountable for their own work. They could get away with saying, I tested the way you told me to test, that's enough.

Whereas, if what you tell them is, give me something that's fit for testing, you know how to do that, we're not going to tell you. That's a very different message than, give us something that passes this suite of functional tests. really Let me rephrase that question for the film. What can testers do to help the development staff generate better code in tests.

One answer is, if the programming group decides to experiment with Watts Humphrey's PSP or with any other process that has them tracking their work in order to improve it, a key thing the test group could do is to support the right of the individual programmers to keep their work products private. There's a wonderful book by Robert Austin called <u>Measuring and Managing Performance in Organizations</u> by Dorsett House Publishing, Austin is referenced both in <u>Lessons Learned in Software Testing</u> and a little bit in the measurement discussions in these course notes.

Bob Austin talks about measurement dysfunction and the distinction between information only measurements and motivational measurements. Information only measurements are things that tell you how you are doing but in a way that doesn't allow anybody to manipulate your performance based on them. Motivational measurements are measurements that eventually land in the hands of for example of managers who can use them to get you raises, give you unpaid vacations, etc. Motivational measurements are more likely to distort what's going on in an organization. Sometimes for the better, that is a result of a change. Sometimes for the worse, in general American workers are resistant to having motivational measurements made of their work. I think for extremely good reasons, but that is my personal opinion. Read more in Austin as he goes through some of the problems with that. But to the extent that somebody is tracking their failures.

Watts Humphrey and I talked about the need to keep private records private. He said, "Don't give anybody your notebook unless you would also be willing to give them a full set of signed blank checks." It's the same level of trust. You are giving the ownership of your job to the person you give that kind of data to. It has to be totally private or available only to someone you trust running your life which is probably not your manager.

One thing we can do to support process improvement by individual programmers in a way that says we will help you keep your stuff private instead of into our metrics empire. Another thing we might do is research some tools they might use if they respect us on what tools are usable. Another thing that we can do is to advocate for testability features in the product. Testability features are features that make a product’s performance more visible and give us more control over aspects of the product.

If we can set the state of internal variables, look if we can check the content of the stack, if we can see more and do more, we can test more. But in designing the interfaces for us that let us create diagnostic tests, quite often we trigger a whole wave of questioning, well what should the value be here, how should I be able to monitor that, and maybe I should write some routines to monitor that myself.

As you encourage people to design for testability, you probably also end up encouraging people to pay attention to the information they are now generating and they might code for better results in that more highly testable code. Even if they don’t code for more better results, you are going to find the problems more cheaply and be able to automate effectively your testing.

Slide 4.14: Defining the Test Approach

We have the mission of the test group, we have the test ideas, and somehow we have to have a mapping that selects from too much stuff to a broad goal. And that mapping is called the test approach and test strategy. The set of selection rules for deciding what test cases to use and what areas to focus on and there are some attributes that are important to that test approach. Now there are some more notes in the notebook and there are lots more notes in the booklet.

Let me just suggest that there are at least five attributes to a good test approach. The first one is diversification. That was what Group 1 wasn't. You diversify because even if you think you know what the primary risk areas are in the program one of the risks is that you are wrong. We start with imperfect knowledge about the product. We gain information as we test. If on the basis of our imperfect information we choose not to test an area at all, or not to use a certain kind of testing technique because we know it will be ineffective because there's nothing it would trip in that area or against that technique, then we'll never discover that we might have been mistaken. That area might have serious problems in itself.

A **diversified** approach is one that covers a lot of different areas, in a lot of different ways. Focused, we spend more of our time in the places we think will give us the best pay off. Best pay off against whatever mission we have, but not exclusively focused. We'll still play around the edges just in case there's a surprise that is an obvious one. There was a comment made earlier today, where if I went through the list, I had the list, and there was something obvious that I could have done and I didn't do it. I came back and said make a list obvious in its breadth too and a certain point, a reasonable manager will say, yes, I understand why you wouldn't do that. On the other hand, there are some bugs that are just so obvious, that if you didn't find them everybody in the organization including the tester sitting to the left and right of you will go, how could you have missed that? And the way you missed that generally is you didn't test in a certain area or even perform the most basic tests in certain area. Don't get lost in that. That's the goal in diversification – to make sure that you are varied enough that it is cheap to find problems that would embarrass you clearly if you missed them.

Risk focus I should have to push too much in an audience of Rational customers because the entire Rational Unified Process is about risk management. The entire iterative development approach is about risk management. We figure out what problems are the problems we most need to manage for this iteration, we run that iteration to do that and we move on. And of course, if we're taking on the programming side, the risk is that we that don't have a certain feature or don't have it in a way that is well, it's the most important thing and needs to be there.

In an infinite population of tests, if you aren't asking yourself the question, which parts of the program scare me the most, which kinds of the problems are the ones that would be most acceptable to the customers, how could I find those – then you will thrash. If all you do is randomly sample, 10% of infinity is infinity, right? If we have an infinitely large population of tests, any sample we could actually achieve, 100K tests, 1M tests, is a grain of sand on the beach compared to the total number tests we could run. If we randomly sweep a few pieces of sand from the beach, we're not likely to find problems that we find. We have to focus. The tiny amount of rework that we have has to be focused on the things on our most-afraid list, we won't get there by luck.

There are a lot of canned test strategies. We'll talk later about test documentation and templates for test documentation that tell you the one true way to define your test documents. It just doesn't work. Different **products** have different risks, different project teams have different risks. Some project teams handle some problems brilliantly and are weak in other areas that other project teams are very strong in. You're not going to worry very much about whether it's a lot of fun to fly a Boeing 727, but if you're flying a flight simulator computer game, being not fun is worse than crashing. If you were to adopt the same strategy for the flight simulator game and the flight, that would be horrible. The strategy has to reflect the problem, the development group, the mission for the product.

Practicality is another important question. Imagine running a test organization where every person in the test group is not a programmer. There are a lot of test groups where that's true. Then imagine telling them that you'll give them the source code, you'll give them j unit, you'll give them automated test generators, and they should go off and do the unit testing for the programmers. This is not a strategy that is likely to succeed. It will look wonderful on paper, but you better fire all your testers and the people who hired them. That group probably has other expertise like a deep understanding of the use case, or is at least capable to develop that expertise.

If you've hired well, there probably very capable of developing that expertise even though they couldn't plug their way out of a loop. On the other hand, if what you found are a bunch of fresh college programmers wanna be's, who came into the test group as their entry into the rest of the company but really didn't want to be testing in the first place, they're bright, they're enthusiastic, they're hard working, they live and breathe code, and you tell them let me teach you about black box testing, you may as well bring pillows because they're gone. Give them some good automated test tools and have them interview SME's and have them figure out as their task, how to automate tests associated with those customer stories, and you might find some very excellent work that the first group couldn't do. This group can't do the first group's work and the first group can't do this group's work.

If you're designing a test strategy that doesn't take into account who your test are, what you can find on your market, what tools you actually have in-house and what you can afford, what resources you have in-house, then you don't have a test strategy, you have a fantasy. It has to work in your environment. It has to be clearly enough thought that when the VP in charge of saying that your testing strategy is stupid, comes up to you and says, "Explain your testing strategy to me, I'm waiting," you can explain it. And when you say, "I don't know why anyone would do that," you can explain that.

And it's not just in this adversarial case. Three months from now the product will be out on the market and surprises will have happened, the product will in some ways fail that no one expected, and people will come back to you and say, why did you do that. Maybe you need to explain.

Three weeks from now you're going to go up to an executive and say, "I need more testers." And they will say, "OK, what are you doing with the testers you have now?" If you merely give them details – these are doing this and these are doing this – the executive, if they have any level of sophistication at all, will say, "OK, there are an infinite number of tests possible and I'm hearing that the test group is choosing a subset of the infinite number of tests and wants to choose a bigger subset of tests. It's always nice to do more, why? Why do we need to do more this time? What's the added benefit from the extra staff? And how do I know that you are using the staff you have today in choosing the optimal subset given your resources?"

That's the issue of **defensibility**. If you want to add staff and you have a rational executive who is budgeting, they need to know that there will be an incremental increase in quality or an incremental decrease in their ongoing costs risks for this product. If you don't have that, you don't have a defensible test strategy. If you don't have that, you probably won't get the staff.

This defines a **context driven** approach. Somebody commented before, "How come we don't talk about context driven in testing?" This is the notion of context driven test plan. Instead of coming up with the one general approach that guide test strategy development always, we end up saying there will be different test plans, different strategies, different sets of documentation, different tradeoffs to get made for every different project. That's just how it is.

It may also be per iteration, you're going to think things through again per iteration. But your overall strategy might last a lot longer than one iteration. There will probably more fundamental differences across products than across iterations. The flight simulator and the plane are more different from each other during the first week of development on the plane than the last week of development on the plane.

Slide 4.15: Heuristics for Evaluating Testing Approach

Given that we don't have a pat set of answers for what a test plan should be or for what a test strategy should be, we end up coming through it a different way. What are some good ideas for test strategies and some good ideas for evaluating test strategies, so one of the pieces in Lessons Learned that we reference in the slides is a collection of test heuristics. There's a chapter at the end of Lessons Learned about developing testing strategies. We cut down the number of slides specifically deciding that it would be easier to work with by just going through the chapter on test strategy in the book.

Here's an example of one of the heuristics we suggest. Optimize your testing to find important problems fast instead of optimizing your testing to find all problems with equal urgency. Now, you saw that test strategy reflected in both Group 1 and Group 2's testing where I said the first thing both groups did was to walk every feature and look for obvious problems. For some folks, that is naturally what you would do. For other folks, they would concentrate on function 1, then go to function 2, then go to function 3 or they would concentrate on the highest risk function, then the second highest risk function or the best specified function and the second best specified function. If functions 1, 2, 3 were specified and were still waiting for specs on 4, 5 some groups would not touch 4 and 5. Some groups will say, it's code, it's in my hands, I will touch it. If something really horrible happens, I can give it back to them and they can start working on it even before I get the spec. Maybe that would be a good thing, maybe when it finally comes to me specified it will come with a spec that actually means something.

In Bach's view and mine, you want to make sure that you're looking for big things early and looking broadly for big things early. And after you've scraped off that first broad view layer, you ask the question, what are the things I want to most look for things now instead of asking the question, how can I most cover the product. Now, you can imagine that under some circumstances that "how can I most cover the product" may be the most important question. Heuristics are heuristics. They are rules of thumb. They are suggestions that are usually useful but not always right.

We provide with the heuristics to test approach, a serious of assertions with some backing some statements on what to think about, things you want to think about but not necessarily follow. You might reject one, you might reject all sixteen for the project. But rejecting them consciously, no I don't want to follow this heuristic, is probably a wiser approach than just not thinking about it.

Slide 4.17: What Test Documentation Should You Use?

Let me hit another aspect of mission. That’s the mission for the test documentation. There are several different questions we’re going to ask in this section. This is an overview slide. We’re going to start with by thinking about the notion of test planning, test documentation standards and templates.

Slide 4.18: IEEE Standard 829 for Software Test Documentation

We’re going to focus on IEEE standard 829, which is really the ancestor of most of the templates you’ve seen. And then we’re going to ask some harsh and critical questions about when these are cost effective. As with the other mission question that we hit, the question about mission for test documentation is not a real easy question. It’s not like, oh, yeah, this is always effective, or oh, no, this is always bad. We have a congruence problem again. Sadly, none of the standards that I’ve seen for test documentation take context into account at all. And so the discussion in the literature on when one technique is more or less appropriate are weak. The Kaner, Bach, Petticord book tries to pick up some of that material. Some of that material is well covered there and some of it is still just published in conference talks. OTOH, a great deal of this is unpublished, word of mouth, discussions in conferences, the type of material that has been collected for these course slides and this notebook and as you think through some of the issues that were raised, you will hit more of them. This is certainly not a complete thinking through of the problem.

It’s certainly a more complex problem than simply coming up with a template. There’s template you can get with RUP that follows standard 829. How many of you have seen test plan templates. They ask for the project plan, they ask you to layout things you are not testing and why, they ask for the test environment, they ask you to define the overall specification for testing the overall strategy, they ask you to define all the items or areas that are to be tested and then they ask for details of the testing. These are all 829 basics for templates. Many of them have gone through six or seven generations of copiers. They’ve been plagiarized by somebody, who plagiarized somebody and probably didn’t even realize it they were taking it from standard 829. so most people have not heard of that standard.

IEEE 829 is where all that comes from. It was an extremely carefully thought out standard that was published in 1983. Every type of information that standard 829 asks for is useful information. If you had infinite time, you would undoubtedly want to supply all of it.

Slide 4.19: Considerations for IEEE 829

But we have an interesting problem. If I actually give you all this information for a single test case, and if I can fit this on one page, I'm a pretty concise writer or the test case is trivial. *How long does it take to write up a test description?* I used to manage documentation groups and I used to have tech writers tell me it took them an hour to write a page. And it did take them an hour a page to write on average if they had done all of their research already, and had done all of their outlining already, and were merely sitting down at the computer to type what they already new they wanted to say. But as soon as you started taking into account, the investigation time, the outlining time, the thinking about what you're going to say time, the checking whether it's right or not, the editing, but also the fact checking, it came down to more like 8 hours a page.

Joanne Hackos in her book on managing technical documentation projects (referenced in the notes), ran extensive studies while she was running her own technical writing consulting company. Her company did tech writing and tracked metrics and she also did extensive studies as the president of the Society for Technical Communications gathering all sorts of data from other companies. And the data that we came up with in the companies that I was at were merely consistent with the numbers Joanne Hackos was already teaching. It wasn't surprising that they were consistent, a whole lot of other tech writing departments were also finding numbers consistent with Joanne Hackos.

So think about this. 8 hours a page for tech writing by professional writers. I don't think testers are going to write faster than people who are paid to write fast for a living. I also don't think we're likely to investigate faster than tech writers. Good tech writers are pretty good at this.

So if this is only a page, then if we counted all of the costs associated with generating this instead of merely the writing time, it will probably be a day per page. But let's pretend that we get the information magically implanted in our heads or that reading specifications and doing stuff is something we can do at night off company time, and take ourselves down to an hour a page. And let's take that we want to support some moderately high volume testing, maybe we'd like to support a group that does automated testing and has a lot of tests.

10,000 tests is not an outrageous number of tests. But if we're talking about 10,000 tester hours of test documentation, if all they ever did in their lives was write test documentation, that would be 5 tester years, more likely 8 or 9 tester years worth of work that you've just tacked onto your project. If you're in a project where you are designing aircraft, then "No problem, spend the time." If you are in a project where you have a client, you're doing custom engineering and the client says, "I'd like to buy some IEEE 829 standard test documentation as part of my purchase." Then you say, "No problem, I sell labor and products and my time for doing time test documentation is this much including my overhead, I'll make this profit on it, if you'd like to have 9 testing years worth, great, no problem. Would you like that on white paper or yellow paper?"

However, on some other projects, those 9 tester years are the difference between getting done and not. It's just not that sensible to add that level of burden. In fact, for many high volume test strategies, the documentation costs would be higher than the cost of creating the test cases.

The key point to think about is there is cost whether f you use the tool or not, the information takes time to collect. And if you have tools that make it cheaper than the first question that you can ask IEEE 829 gets answered a little differently, what's the documentation cost per test case?

If we want to evaluate this standard, one way to evaluate the standard is to evaluate it the way they do it in the way they do in the Software Engineering Body of Knowledge (SWEBOK) that the IEEE is publishing now that basically says this standard is good for all purposes. If you're going to turn your brain off, no problem.

But if your goal is to help your Project Manager get out the right product, with the right quality, on time and within budget, then you face tradeoffs. And if you're going to trade time on this against time finding problems or against time helping people improve other aspects of their process, then it's worth taking a very careful look at your costs and benefits associated with doing this and so we end up with some cost questions. How much does it cost? And not just to write it but to maintain it, if you change the design, how much do you change these very detailed specs for test cases.

Do we end up with the situation where at the end of the project, the Marketing Manager runs screaming in and says, I need you to change the project and the test organization says, we can't; it will cost too much to change the test documentation. This happens. You have to decide if that's what you want to have happen in your organization.

If you think you're doing iterative development, then in every iteration you need room for change, including room to rethink what you did last time. And to the extent that anything you do creates a maintenance cost, if you change then you'll have to change this and it'll cost you money, then you are creating inertia. You are creating resistance to change. Every time you look at an iterative project and say, "I'm going to build something now that downstream will create resistance to change," you are creating a less iterative process and one that is more like a waterfall process. Changes made early create high taxes, if you want to change those decision later. Now it might be worth it to add that inertia, this isn't tell you it's not; it's telling you to think about it. How much inertia does your process want to afford?

Another kind of inertia that I mentioned relates to what I'm calling invariant regression testing. If you're documenting every little bit of your test case than you're creating a price that might discourage people from variation for reasons other than the value of the bugs that might be found or the difficulty in coding a more flexible solution. You are adding a price associated with the paperwork and that price might not be where you have the decision point for whether you want to add variation or not. Anything that blocks you against change, is a cost against change.

Slide 4.20: Requirements for Test Documentation

Now, standard 829 in my experience leads to fairly inflexible testing strategies. It was developed with the assumption of a non-iterative waterfall process. It’s used in my experience almost exclusively in groups that want to do waterfall based development. It can be applied elsewhere but rather than starting from saying, I want to use this. It’s like someone comes up to you and says, I need to write a program. And the first answer you give is, use COBOL. It’s an answer, it has some costs and benefits, but it might not be (it might be), but it might not be the best solution for their circumstance. You might need to ask why do you need write this program, what kinds of things do you want to do before telling them a detailed solution. 829 reflects a detailed solution.

Slide 4.21: Test Docs Requirements Questions

Lessons Learned has a lot of requirement oriented questions. Here's an example. Is the documentation you are about to provide a product or a tool? Here's an example of the product. You are building a telephony system. The people who will resell your telephony system, you are a small PBX developer or a small system developer, and the people who will market your system are large telephone companies. They will assume that they're going to have to take care of the maintenance and the software if you go out of business. All sorts of telephone equipment comes with warranties that last 10-25 years and all sorts of companies that make these components don't.

So if you're writing software for a product that must work well, like telephone software, and will be supported for 10-25 years, then the company that will take over the resell and maintenance for this has a very strong interest getting very good test documentation. Because if they don't understand how to change your software and fix it after you're long gone, they can't afford to take over the sale and then maintenance of the product. In that world if you're not giving them documentation like the 829 documentation, you are probably making a horrible mistake. What you really have in this situation is you are selling the ability to maintain your software to somebody who has no real access to the original development system or original development thinking, and if you're not selling them with that extensive documentation they probably cannot afford the risk of buying your product.

Many complex products that are done under contract must be documented that way. This is part of the reason why DOD standards look very much like 829. You end up – are we going to build an aircraft, great – we're going to hire a contracting company to design the aircraft and by the way this is contract, when we ask them to do some maintenance later, we may ask the same company and we may not, and they may have the same development team and they may not, we aren't going to have any control on who they are going to put on the project this time and if we don't have great documentation to pass right back to the next team that's going to work on whatever it is they're going to work on, they're going to bill us through the nose for all the reverse engineering time.

So if we're DOD, we're going to demand as part of the cost of the product up front when we get the contract for thorough enough documentation that as we go along the next 20 years we can keep passing this information along. As the company that's making the product, we have a customer who says, I need documentation of this quality, we say sure, no problem, we have that. Now contrast this with a company that's making a word processor for the mass market. Your word processing customers don't care. You guys are all word processor customers – how many of you have inspected the test plan for the Microsoft Word?

Yes, there is a test plan. But we don't look at that. To the extent that we still have a choice in the market, the question isn't what's the test plan – that's not what is going to guide our purchase of the thing. Those are the kinds of things we're going to think about. It doesn't matter if it's been thoroughly tested or not thoroughly tested. A thoroughly tested product that's going to get us a macro virus everyday is undesirable compared to a less well tested product that just doesn't have certain kinds of problems that we don't want to have.

So we evaluate from the outside, what does it do for me? Not from the process side of how well tested was it – who cares – it's how well it works not how well it was tested.

So in that case the test documentation is a tool, it's not part of the product. It's part of the development team's arsenal for building a great product. In that case, the expenditure we want to make for this product, this subproduct – this set of test documents – is the minimum we need to be able to develop the end product we need to sell at the quality level that we need. Anything beyond that is waste. Any level of documentation beyond what we need to generate the product at the level of quality we want, is waste. Whereas when we have a customer saying, I want it this way. Even if we go way beyond the minimum we need to develop first release of the product in good shape, way, way beyond, if they understand their business reasons for wanting that extra level of detail, then we're giving them a product. It's not our waste, and if their business analysis is correct, it's not their waste either.

If we're only thinking about this, 829 might be a good documentation standard for delivering the tool depending on what the customer needs. You might be able to use 829 to help you and the customer define what he needs. It is a very effective tool for that purpose. It's kind of a shopping list of the kinds of documentation you need and then creating the documentation to match the shopping list. If you're developing something in-house, the message might be to select from 829 those things that are useful, or the message might be to shred 829, or the message might be to still use 829 to the extent that it protects you from some other risks like remediation.

But to the extent that you are publishing a product that someone else will pay for, publishing documentation someone else will pay for that, a standard like 829 is probably a useful structure for that publication.

Slide 4.22: Write a Purpose Statement for Test Documentation

Your test documentation has to support the mission that you had for testing. And so, let me give you two examples of mission statements for test documentation to give you an illustration of the difference. After you go through trying to figure out what your requirements are and the test requirements questions in my experience as a consultant are handy for doing that for interviewing a client and asking what are you really trying to achieve.

At some point you write a statement that preferably fits in a paragraph, that is short, preferably, but not necessarily is one sentence. The main thing is not whether you can fit it into one sentence, but whether it has more than three components. The more complex it is, the less meaningful it is.

So here we have the first one.

The test documentation set will primarily support our efforts to find bugs in this version, to delegate work, and to track status.

For mass market commercial software, that's almost always the level of documentation I'm looking for, I don't know if I'm going to have another version, I don't know how much change there will be between this version and the next version but it will probably be substantial. I have the problem of the staff coming and going in my project and so I need to come up with ways to, as with the matrix for example, ways to use new people very efficiently, how to understand what they gotten done and how to report back to management this is what I had in mind to be done and this is how far we gotten. If I can get all that from my test documentation, I'm a really happy camper.

Now let me describe something that reflected much more the experience I had working in a telephone company, a PBX manufacturer.

The test documentation set will support ongoing product and test maintenance for at least then years and will provide training material for new group members, and will create archives suitable for regulatory or litigation use.

For that we want a standard set of materials. We want something that is going to pass from generation to generation to generation of tester. By the end of 10 years it is likely that the last person who worked on the original test documentation set will not be available to talk to the most recently hired person 10 years later. There will be a complete turnover of people in the chain.

So the only record for many readers will be the written document, this is what we had in mind. If that's where you are and you really believe you're going to making software changes and therefore doing extensive testing 10 years later, now it's worth investing in the lifetime value of the product. If you think you are making something that will be obsolete in a year, the investment is very different.

Slide 4.23: Exercise 4.4: Purpose for Your Test Documentation?

Regroup into your project teams and take ten minutes to discuss the exercise. Write down the answer, so you can share it with the group.

- Use the company and product from Ex. 4.3
- Reform project teams
- What's the test documentation mission

Module 5: Test & Evaluate

Slide 5.2: Module 5 Agenda

Module 5's basic workflow is test and evaluate. This is really the workhorse workflow and we're going to spend two different modules on this workflow.

If we think about this module itself, what we're going to focus on are the primary types/styles of functional testing. The next module we're going to focus on problems reports, change requests.

Slide 5.3-5: Workflow: Test and Evaluate

Then within the workflow itself, basically the point of the workflow is to look at the breadth and depth of testing and how it’s done.

And, so in each cycle of testing you have a build, you take that build and you subject it to a bunch of tests and you’re trying to figure out what’s wrong with that software and how we can talk about it in ways that might get it fixed.

So there are a variety of test activities that you look at in rough, basically you’re defining tests, running tests, finding problems, logging problems, and advocating quality.

And the usual cast of characters are doing all the activities. I have those activities defined and characters defined in the notes. They produce the predictable kinds of artifacts that are also defined in the notes like test results, a summary of what actually happened during testing, test data the data you use to feed the test cases, test cases themselves, this is what we mean and how we define the stuff that we do against the program one piece at a time to find bugs, and … You can go to RUP itself to get more detail but we talk about test cases.

Slide 5.8: Discussion Exercise 5.1: Test Techniques

Test cases really in a lot of ways are generated according to your notion of what testing technique you want to use. And one of Kaner's grad students did an inventory recently across literature of the number of things that have been identified as distinctly published test techniques. She found about 200 of them. They all sounded the same.

So, here are three examples that might help us understand when people say, oh I've got a new test technique how the focus of different test techniques might differ.

Can anyone tell me what user testing is?

OK, testing that involves the end user.

When you think about end user testing, are you thinking of how that test would be done?

No. Doesn't specify.

What things would we test – is that specified?

No. Doesn't specify.

The key thing about that term (user testing) is it's focused on who does the testing. It is not focused how they test, what they test, when they test, how to figure out whether or not the product is right or wrong, the key thing that we know when somebody says – we're in user test now – is that there is a certain class of humans that's involved with the product in some way.

Usability testing is not necessarily the same. What is usability testing?

Ease of use.

Okay, you're looking at ease of use. Any other comments on usability testing? Okay. Accessibility testing may be a subset of usability testing. Okay, fitness for use. Okay, so if it's a high volume, like a data entry thing. Productivity type of metrics that would go along with it. Okay, productivity, efficiency of the user. . . I mean when people start how many mouse clicks does it take to do this task, it takes 47 but I want to do it 3. How angry people get using it, how easy it is to learn, how happy they are playing with the program.

All of this kind of stuff I want to suggest focuses on a *class of problems.* The class of problems that are – this really might not be usable. This might not be accessible. This might not be any fun. This might waste my time.

Now if I told you we were doing usability testing, would that tell you who's doing the testing? We might be doing it with normal humans, on the other hand we might be doing it with testers? We might be doing this with experts, we might be doing this with anybody as long as the information they can give us back will be information about whether in the hands of the people we want to have it in the hands of, the result will be a useful result.

And what about user interface testing? What is that?

User interface, check it against standards.

Try every damn thing I can do with this combo box to make sure it doesn't give me a bad event. Try all the controls. User interface testing generally refers to what I want to call coverage. Here are all the little elements of the user interface and we better check each one. Now when I'm doing user interface testing, I'm more focused on making sure that every piece is there and works in the expected way that conforms to some standards if they exist than I am about who is clicking the box to see.

The risks I'm looking at are really the risks that this thing doesn't work in the normal way. I'm not driving the testing by saying, "Hmmmm, what would unusuability mean? What sort of problems are usability problems?" I'm not coming in with a class of, "Gee these are failures -- let's try to figure out what things we'd test find out what is weak in this area." Instead, I'm saying, "Let's look at all the things and see if they're weak."

Here we have three commonly talked about techniques, user testing which is nothing more than a certain class of people we're going to subject to our program and see if they can use it, usability testing – here's a class of problems we're going to try to find out about, user interface testing – here's a class of things we're going to check out.

Slide 5.9: Dimensions of Test Techniques

In general, when people describe techniques, they describe things that focus on one or two or three of these five dimensions.

1. Who's the tester?
2. What are we testing?
3. Why are we testing it – what kind of problems are we looking for?
4. How do we test it? An example of how we test it – GUI regression testing – who's the user, well the machine but who's the person driving the GUI regression test, well somebody. What are we testing? Whatever we can reach with the GUI regression test tool. What problems are we looking for – whatever we can find with the GUI regression test too.
5. Evaluation – how this particular test tool works given the level of custom controls and standard controls that are available?

People constantly talk about it, "The technique I'm using is gui regression testing using tool X."

Then we have evaluation based techniques and so for example, there are tests that are based on oracle. I'm not talking about Larry Ellis and the Oracle program. I'm talking about the old Greek oracle you'd walk up to and say, what's the truth? She'd say, 72. You'd go, "Hmmmmmm, what does my program say the truth is – 72 – ack, it passes!"

If you have an oracle, a reference program that can let you know if your program is giving you correct data or not, then you can write massive automation series. Basically driving your program, driving the oracle and saying what's the answer, what's the answer, they're the same, what's the answer, what's the answer, they're the same, what's the answer, what's the answer, they're different – call a human, something's broken. It might be the oracle, it might be the program under test. The oracle is a program you trust more than the program under test to give you the right answer. Doesn't mean it's perfect, it means it's more perfect that the software under test. If you have an oracle, you can do a lot of testing without human intervention whereas if you don't have an oracle, every time you run a test, you have to call the human over and say, what do you think. "Is that right or no?"

It's still not fully automated. But if you have an oracle available, then the execution tools that you guys sell, you can use them brilliantly to achieve very high volume testing, very thorough testing of some parts, anything you can evaluate, some parts of the system. And if we look at oracle based testing, who's doing the testing? Well it could be anybody, anybody who understands how to use the oracle and how to use the tool.

What gets tested? Whatever is in the scope of the oracle. What are the potential problems? Anything that we could detect by comparison against the reference program. That, by the way, is an important piece. Let's suppose we have a program that we know adds correctly, and we have a program under test and it's supposed to add correctly. Two spreadsheets, for example. One is perfect and the other is the latest version of a spreadsheet that you've been coding for yourself.

So you add two cells. You add 2 + 3 and the oracle says 5 and your program says 5 – you suppose it passed. But what if your program went, "Hmmmm, hmmmm, wait five hours hmmmm, 5," the oracle looks at the answer 5 and supposes your program passed. But any human at this point would have said, "Nah, something really bad is happening with your program!"

All oracles are partial. We still have a lot of room for human observation and human judgment. And if you do very high volume testing based on the oracle, the only things you can detect are those dimensions that you can compare from your program to the oracle. Everything else is an unmanaged risk. That is to say a risk that has to be managed by someone in formal testing. It's not a complete automation and it's not a complete technique in that when you do this, you don't have to do anything else.

Many techniques really can be defined as one dimensional, other techniques might be defined as 2-3 dimensional, but just about anything that anybody names as a technique could be characterized along one of these five dimensions. And when you finally get around to applying a technique to an actual situation, then you have to specify all five things. Somebody has to do the test. They better evaluate the results are they are not testing. They are testing something, if their brain's engaged, they are looking for some problems and they are testing it in some way.

And when we finally get down to it, we specify all five things but the most general techniques specify one and leave the other four or three undefined. Which is why many organizations will do testing of a certain kind, all we do is user testing, and yet achieve a very thorough, evaluation of the program. All we do is user testing but the all we do turns out to test everything from many, many different angles in lots of different ways. Somehow we've found a user community that's really invested in – we're doing in-house development – the user community hates the programming organization and they've set up some really thorough tests before they accept this software being applied to them and so, yeah, it's user testing but it's the kind of user testing you pay a test lab to do and be happy. OTOH, other user testing might be very literal.

Is oracle a widely used term?

The term "oracle" is used widely in two slightly different ways. The way that I've been using it is the narrower sense -- an oracle is a *reference* function: a function that generates a value that you can compare to your program's value with these to tell whether the program has passed or failed the test.

The broader use is that an "oracle" is an *evaluation* function which will tell you whether the program has passed or fail the test. In my use of the terminology, I have both an evaluation function and a reference function, so when I have software in test I have something I can use to tell me whether the program passed or failed. Very often that something is a comparison to the output of some other program; say, "Do these match?" So I have an evaluation that checks an oracle or I have an evaluation that doesn't check an oracle but checks something else.

(BTW, the odd thing about the Greek oracle even though we use the word is that you look at what the oracle actually said back in Greece - it was always widely ambiguous. It might turn out to have been the truth but you know you had a series of horrible tragedies that occurred because people misinterpreted with the oracle meant so…)

Slide 5.10: Test Techniques—Dominant Test Approaches

Having hit the general notion of a test technique, which I say is basically a five dimensional thing, the next piece that I'm going to take this to is 10 wildly different test techniques that I think are the dominant testing techniques.

Well, here are bunch of what you could call techniques. Sometimes they are called testing paradigms and are sometimes called testing styles.

So if you were to go around a wide range of companies, what you would find is that some companies would completely focus on function testing, that's all they do before releasing a product. Some would focus only on risk based testing, some would focus only on stress testing, and so forth.

For each of these, you could probably find one or several companies that adopt one of these types as their dominant style of testing. They might do a little of a different style, but when you talk to them about their style of testing, they will talk in terms of one of these approaches.

When I was a less experienced consultant, I had my favorite two approaches, I knew those were the two true approaches. If you did these, equivalence analysis and scenario testing, then that was good enough. And it took me along time to learn that clients of mine were finding problems much more easily than I could with my techniques because they were using other techniques. I would look and say, this is strange stuff to do for testing – why don't you take my ideas. They would take my ideas and get better because made them diversify.

It took me awhile to understand that diversification should work both ways. That I should learn the other techniques as well because there were a lot of things that I would learn how to do cheaply that I didn't know yet how to do. Very few companies operate with more than 2 or 3 of these techniques as prime things that they pride themselves on practicing.

The best thing that you might take out of this entire course, is take one technique that your company doesn't do very much, take just one – don't make too many changes at the same time – and add it to the list of things that you really drill in. Get really good at it in six months, then add one more.

Slide 5.13: Module 5 Agenda

What we're going to do for the next many slides, is to look at each of these general approaches in isolation and to say, imagine we were in a company that lived and died by function testing. What would that feel like, what would we do, what are they looking for, what are the strengths of that approach and what are the weaknesses of that approach. Similarly for random testing and domain testing and so forth.

Slide 5.14: Test Techniques—Function Testing

Let’s start with function testing. The general notion of function testing is, is it’s black box unit testing. It is testing each function one at a time, usually people talk about you confirm that it works. If it is the only style of testing that you do than you probably test every variable in that function pretty harshly. You look at anything within this one but you aren’t looking at interactions among functions and saying here’s a feature, here’s a feature, here’s a feature, here’s a feature, test this one, test this one.

You check off a function list and once you finish the function list you say you’re done. And your belief is if all the pieces work, it will all work together. It’s not always true, but that is the belief of the people that operate in this style. Now, many, many companies adopt this as a secondary style. When I say secondary style, many companies use this in a less extreme way to qualify the program for testing.

It is a good practice to focus on function testing first is a project risk management approach. In your structure you can assure yourself that testing of a certain kind won’t block you and delay the discovery of certain kinds of problems. That's the only reason people start with functions first, or with individual things (whatever the unit is), to avoid blocks.

Skills involved

Please think about the mission exercise we did (Slides 4.6, 4.8). The first thing that Group 1 and Group 2 did was function testing. Let’s check A, B, C, D, E but that’s not what they finished with. They probably tested A in lots and lots of ways, and may well have tested A in conjunction with B, C, D, E but the first thing they did was wander through and say, is there anything obviously wrong with this one, anything obviously wrong with that one, anything obviously wrong with this one. And it’s important to start your testing by testing the items in isolation because early in the program’s life with you, it probably doesn’t work in a lot of fundamental ways. If you come in with really complex tests, all you get when the program fails is a really complex troubleshooting job. It’s much faster, much simpler to come in and say, how about this, how about this, how about this, and qualify the program to be worth doing combination testing.

The core tasks of function testing involve isolating what the functions are. Now I note these in the student notes, not the instructor notes, the student notes and your course hand out.

And so you can refer to those, you can say look, if you were to practice this at home, the core thing you would be doing, the first thing that you would be doing, is going through the program and asking yourself what are all the features, what are all the benefits, what are all the variables, what are all the individual things that I can test and make a massive list of them - this is often called a function list. I don't think a function list was ever really just a list of functions in the program; it’s a list of all of the things that can be manipulated by a tester in a program.

Then you ask for each one - how can I test it on its own. And you might use boundary techniques for one variable at a time. You might use use cases for one function at a time but you're not going to use complex use cases that will carry you across several functions. You are going to use the happy path and as many alternate paths as you can think of this individual function.

Now is the function the thing inside the code that is labeled function? No you can’t see that. It's a stand-alone piece of functionality that you notice as a tester. In recognizing those many folks are pretty blind to all the pieces of functionality they end up interacting with in the test program. But the first skill you develop as a functional tester is the ability to walk the program and build that list.

Take home exercise (in student manual)

An exercise for you is take part of a program that people know like Microsoft Word.

Have people work in pairs, it is always in my experience more effective to have people work in pairs, have two people sit down at one machine and start making notes and talking about what they see and crank a list out.

Then have folks compare their list - everybody's working on the same function. Pass the lists around – 0h, gee what did you get, what did you get? If you take a small piece, it might be a fair task to have people work on this over an hour. (If you take something smaller, like Notepad, they can do this in much less time. But you're not going to have the rich variation of experience if you do a trivial program.)

What you find when you have something that is complex when you do this exercise is that different groups will do a creatively interesting job and they'll have places where they don't overlap. This group does a really great job on these things and they have blind spots. This group does a really great job on these things and they have blind spots. And you end up saying if you're going to get skilled at this you have to be building for yourself a list of categories of things to look for that you can then break down - here's another one of these, here's another one of these, here's another one of these, here's another one of these, here's another one of these. Having something complex enough for people to compare notes and really see differences is an important part of the learning experience.

Although in a two-day seminar the hour that it could take, plus the debriefing time - debriefing isn’t necessarily everybody stand up and do a presentation in the class whilst everybody else in the class falls to sleep. But debriefing where everybody photocopies their notes and passes them around or debriefing where you have flip charts and everybody having basically having finished their flip charts walks to everybody else's flip charts and goes oh, gee so that's what you did, can both be interesting ways of folks understating what they got and what they didn't. At that point you pretty well exhausted function testing.

Slide 5.17-19: Test Techniques—Equivalence Analysis

The next approach is equivalence analysis testing and there are a bunch of different names for this.

(Sometimes people call equivalence class analysis "domain testing". "Domain testing" is not a good name to use in the context of Rational, because domain is an overburdened word and it has other uses in your process. But people call it domain testing because they think of a function as having a domain, input space, and a range beyond the input space and so your focus is on input variables and all the variables you think it can have, people call that domain testing.)

Stratified sampling

The core idea in equivalence based testing is the recognition that there are way too many tests to run. We just can't run them all.

So, here we have this universe of too many tests and the strategy we follow is to divide it into a bunch of equivalence classes and then to sample it, one or two values for each class. How many of you have every wondered how they do a Gallup poll? I see several of you nodding. The Gallup poll involves something called stratified sampling. These people can call up 2000 people across the US and predict with some accuracy the results of the election.

It's not a random sample. They subdivide the population into equivalence classes, and they say – we can represent this portion of the American population with this one person. We can assume that this one person will vote the same way as everybody in this class. This person is a good representative of that class and our classes are fairly complex. It's not just people who make lots of money, people who make a fair amount of money, people who don't make quite as much, and people who really should make a lot more. That's one dimension, but we also have where people live, what their gender is, what their age is, what their race is, and what kind of car they drive as other variables. But we end up picking somebody who is a point on many different places – this kind of car, that age, and so forth, and we say they represent a bunch of other people who have this kind of car or this kind of income group, and so forth.

They're dividing the world 3 or 4 or 5 dimensionally, but they still end up with equivalence classes. Where they say there are a whole lot of people who have the same kind of income, drive the same kind of car, and basically live in the same kind of neighborhood, and who have basically the same kind of sexual preference. And any one of them could speak for the group, we just hope we get a "most" typical representative. The one who would vote the way most of them would vote. And then they call up their list of 2000 great representatives and weight them according to how often that subgroup fits into the population and then predict on what these folks say what the whole subgroup would do. They actually take more than one representative from each subgroup just in case. That's called stratified sampling. You divide your population into different strata, into different layers, and you make sure you sample from each one. We're doing stratified sampling when we do equivalence class analysis.

These strata are just equivalence classes. The core difference between testing and Gallup-poll-type sampling is that, when we pick somebody in this case, we're not looking for the test case that is most like everybody else, *we're looking for the one most likely to show a failure.*

Think about the example we looked at in Exercise 2.2, the integer group from 20-50. Now 25-30, they're okay, they're all members of the two digits between 20 and 50 the computer should like them, equivalence class. They should all work. But 20 is a little more interesting because if a programmer somehow misspecifies, or whoever does the specifying misspecifies, the program miscoded or somebody misspecified the lower bound, maybe it will reject 20. And we wouldn't see that on 21 or 47. We'll only see that at one place, if the bottom end of the relation is misspecified we'll only see that at 20.

If we know that 20 is as likely to fail as anything else, it doesn't like the two digit numbers, it won't like 20. 20 is as likely to fail as everything else, except it has one other way to fail. Maybe the lower bound is misspecified, then 20 will be the best representative. A little better than any test we could run, because it can fail in all the ways any of the others can fail plus one more risk.

Since we're looking for the greatest efficiency we can get in testing, every little bug we can fit into one stone, kill a bunch of bugs or at least look for a bunch bugs with one stone, that's great. And that's the essence of the equivalence class analysis.

Now historically, we think of boundary cases as best representatives. We think of them as best representatives, because there's that extra little risk in boundaries. But sometimes once we've started thinking about it as a stratified sampling approach, we realize that sometimes the equivalence class is not on a simple straight line.

Printer compatibility

Testing Computer Software, which doesn't come with these course notes, comes with an entire chapter on printer compatibility testing. And the focus of that chapter is how to take what was at that time 1200 different models of Windows compatible printers, which no one could test all of them. No manufacturer software tests all of them. And even if they were willing to try to test all of them, they wouldn't try to test all of them with the average 5 drivers they could work with. And then interacting with all the video cards used on the system, BTW sometimes if you use this video card and that printer, you crash. That has been a kind of failure that has come up with Windows 3.1 and Windows 95 with some specific applications, a conflict between printer and video drivers.

Pretty soon you end up with this enormous number of tests, times another enormous number of tests, and you say, "We can't do that!"

How do you decide what printer compatibility you do? Well, you say 1200 printers – too much – we need equivalence classes.

So for example, there are about 300 printers that are Hewlett Packard LaserJet II compatible. How do we know they're Hewlett Packard LaserJet II compatible? Because their marketing people say so. They advertise, LaserJet II compatible – what could be more trustworthy. Well, some of them might be *more likely to fail than others. But that makes them real good representatives*.

In general, printers that are advertised as LaserJet II compatible, could be tested as a group. Or at least if you're saying I have time to run 40 printers, except to say I need one or two representatives of the LaserJet II's, I need one or two representatives of the LaserJet 4's, one or two representatives of each of the classes of Canon's and so forth, you end up with Postscript level 1 or Postscript level 2 or Postscript level 3.

You have to break them into classes. Are there incompatibilities within classes? Of course. With Postscript level 1 you have for example several different Apple laserwriter printers that are all Postscript level 1 printers, yet they all behave differently from each other. It happens. And Apple has often been talked about as the reference standard for Postscript level 1 compatibility. Yet there's slight variations.

If you're facing 2000 Windows compatible printers, you can't do exhaustive testing. It would be silly to do random testing. You'd end up with an over-proportionate of basically equivalent LaserJet II compatible printers.

Instead you want to stratify and say, "I want one of this group, and one of this group, and one of this group." Now the next question is, how do we find the best representative of the class? Once we've said, let's do LaserJet II compatible printers, where is our best representative. What is a boundary for LaserJet II? What does it mean to have a boundary for LaserJet II? What would it mean to have a boundary for LaserJet II? They don't all fit in a number line; they're all in a box. Anything in this box that says LaserJet II compatible fits.

There's no linear relationship. The way that you operate with that group is through *specific risks*. I'll give you an illustration. Suppose that you're testing an application that makes a substantial use of memory. It will push very complex images through the printer.

The LaserJet II's all had two memory related error messages they could give us, 20 and 21. 20, if I remember correctly, is out of memory and 21 is too complex. I may have those backwards. 21 may be out of memory and 20 may be too complex. Out of memory you can fix by turning the printer on and off and try to print again. Too complex might mean this will never go through the printer. You get a too complex when the printer looks at the image – how does this thing print right, you take this paper, you have the rollers going chug, chug, chug, and the paper says – I'm going through – and you have this laser that's going zap, zap, zap, zap, zap, zap, and the page is rolling while the laser is going. Now if the processor in the printer takes too long to decide whether to zap – oh, yeah, back there I should have zapped – in the meantime the page has rolled. Too late.

If the software in the printer evaluates it and realizes it will not be able to control the laser gun at the right speed, it punts. It just rejects the image and says, I can't print this. Now, the 20 error – too complex – has been the subject of a tremendous amount of work by different manufacturers to use the underlying Canon print engine. It's not really the fault of the Canon print engine, it's the fault of the software that drives it. But, different printer manufacturers have a lot of challenge trying to set up their print software so that they never get a 20 error. Folks who haven't used that engine for some reason, like Lexmark for example who came up with printers that were

LaserJet II compatible but didn't have the risk of 20 errors.

So you have some things that have zero risk that are not worth testing for memory problems, they're not going to fail. They might fail in other ways but not on that dimension. If you look at the 20 problem and you say I want to make sure, here I'm testing my application – say it's a desktop publishing program – I want to make sure that I never get a call from my customer that says, I made an image with your desktop publishing program, I spent hours and hours and hours, I need to print out the brochure and now it will not print. You told me your program was compatible with my printer and I'm angry with you. That's the kind of call you can get with 20 errors.

And so, what you have as the application software vendor at this point is the desire to make sure that whatever you do, you simplify the image enough that it will pass through the printer.

So what printer do you test with? It turns out against that risk, the original LaserJet II is probably the best representative. As they progress beyond the LaserJet II, they kept adding more and more optimizations to fail less and less often.

So if you passed the original LaserJet II you've probably passed all the subsequent LaserJet II printers and if you're looking at the non-Hewlett Packard LaserJets, they either cloned the original LaserJet II software or they cloned the LaserJet II+ which was plus better memory handling or they closed something subsequent. And to the best of my knowledge, there were no printers who were worse on that dimension than the two. There were a lot that were the same, but there weren't any worse.

So as the best representative of this class against the risk of this kind of memory failure, you had the LaserJet II.

But here's another kind of risk. Suppose you had to print at precise locations on the page. Every printed on preprinted forms? It's nice if you can print between the lines instead of crossing over them. But if you're going to line your paper up exactly, especially if you're going to print your form over and over again, consistency in the printer is a very important issue. And that consistency is partially determined by the software, partially determined by the mechanics of the printer, and partially determined by the software of the application instead of the software driving the printer.

So here you are testing the application and saying, are we doing as good a job of achieving consistency as can be done. The HP LaserJet's – it's a waste of time testing them. They've been made as consistent as they can be made, they handle their paper paths just wonderfully. There's a model I used to test for Panasonic that was just a dream for giving the application developer the opportunity to screw up and do inconsistent paper handling when they thought they were doing it consistently. And in my experience, if I could get software to work with that one, it worked with the rest.

Different risks, different ways it could fail, end up driving us toward different specific members of the class.

How do you figure out which member of the class is the best one against a specific risk? Hopefully you have a technician in your company that wears a propeller beanie that is marked "I know printers." Or "I know network cards" or "I know whatever it is" otherwise you're going to have to do more generic testing. But if you really thoroughly know your domain, then what you're going to see is that you've got an equivalence class, a group of printers that work basically the same way, and then a small number that are just slightly more likely to fail in a certain way and when you want to test for that error, you want to represent the class with them.

That's a very real life example of multi-dimensional equivalence class based testing. Still stratified sampling but with this notion of best representative is the one most likely to show a problem if it's there. The more dimensions you work together, the more you have to know about the domain. But this is a very powerful approach for taking massive and intractable conditions and boiling them down to 40 or 60 tasks. Something we can work with instead of thousands or millions.

This particular kind of equivalence analysis based on the printer example is easy to teach. It generalizes well. If they can understand how to do it on one dimension and two dimensions, then theoretically, they can understand how to do it on three or four. Training people how to do it for video cards and modems and so forth, how to get the domain expertise for the stuff they're trying to do takes longer. But people understand why they have to do that and what they need to do. It's just work. It's not conceptually hard to do. And that's made this the dominate form of test planning, when I say dominate, you can read several books on testing that teach only this test technique and test design.

Subject matter expertise

One of the things that I want to highlight is the need for subject matter expertise when you're doing that.

The question came up, I'm not sure when in the sequence of tapes you're going to see a discussion of this group about hiring of software testers, we had a short discussion of that yesterday, but one of the points that came up yesterday was it’s important to staff the test group to boost diversely in terms of theirs skills and backgrounds. If you were selling a product that does printing extensively where the quality of the output is an important part of the benefit to your customer, then you probably want one-person in your group to be an expert in printer output. You don't want your entire group to be experts in printer output but if you don't have at least one expert in printer output, then you will never be able to do printer compatibility testing in a way that can cover the things that you need to cover and free you from spending all of your time installing and uninstalling printers.

The Subject Matter Expert understands what historical patterns of risk have been associated with the domain that they’re an expert within, and that is domain specific knowledge.

And some people understand networks, some people understand printers, some people understand databases, some people understand programming languages, and some people understand more than one. But there is no substitute for expertise when you're going for efficiency. If you're going to risk-based partitions, let's look at a group of tests and say these are similar because they will all fail in the same way. That's really what equivalence class is. If they’re failed, they will all fail in basically the same way. The same trigger will cause all of them to fail if you're going to do risk-based partitioning. Then you are going to say there's a best one for this risk, one that is a tiny bit more vulnerable than the others. You can't know that unless you really understand the risks that are involved. The less you know, the less efficient your partition is going to be.

Blind spots

However, we have some problems with this approach. It’s not a silver bullet. None of these are silver bullets. Let’s suppose in this program that goes from 20 to 50, let’s suppose that there was an optimization at 27, which is the most commonly entered number in this system. At 27 the programmer has cleverly written a new set of code that runs much faster than how everything is handled in 20 to 26 and 28 or 28 to 50. Of course, the optimized code does not work. Well, you are not going to notice optimizations if you always test at the extremes. And you don’t think the programmers are going to tell you about every optimization in the code. Now, hopefully if the programmers have done their own unit level testing, this is an example of the situation where if you see the code, you can see conditions you can never see on the other side.

But values like this, sometimes people say that never happen. People never do those optimizations. There’s even a way of dismissing this among folks who like to do equivalence class based testing, they talk about what’s called the Competent Programmer Hypothesis. And the Competent Programmer Hypothesis boils down to the statement, no competent programmer would make a mistake like that or do something as irresponsible as that, so I don’t have to test for it.

Doug Hoffman’s story

I’m going to close today’s class with an example from the MassPar computer that is talked about by Doug Hoffman who is one of the test team lead on the MassPar. The MassPar stands for massively parallel. This massively parallel computer has 65000 parallel processors. It was designed to be a really fast computer. And what Doug was testing were the integer mathematics functions.

So one of those functions being tested was integer square root. Now, let’s imagine what a test of the integer square root would look like. Any number that you can fit inside the integer is valid. What this means is that any bit pattern between 000 (32 zeros) and 11 (32 ones) is valid. All the bit patterns you can fit into the 32-bit word, can be square rooted by this routine. There’s no such thing as an invalid entry. It looks at one word, it takes whatever is there, it says that’s a number, I’ll take the square root. What tests would you run?

Any bit pattern is interpreted as a number. So if you can specify an alpha, it has a bit pattern and it becomes a number.

So all zeros, it’s a good point, but what gets to the processor is just a pattern of ones and zeros. If you thought you were typing in letters, they get changed to ones and zeros. Yup, I know that, that’s a 111111111 – pretend there are 32 of these – and so just take the square root. You think you’re typing A, it gives you back the square root of whatever bit pattern was associated with that. Now, you guys are tired enough that instead of running this as a discussion, I’ll just suggest a few other cases you might like. Like 0111111 and 1011111, etc. run all the zeros through so that we have one zero and everything else is a 1 and similarly run the ones so that everything else is a zero and have the one run through every possible position. Suppose I did all of these, every pattern whether all zeros, all ones, one zero, but in every position, one one, but in every position. Would that be enough?

Is that all the cases? How many tests of integer square root would you want to do? That gives us 66. Is 66 tests for integer square root enough? Are there any others that come to mind as obvious things you'd want to do?

If you have time and you have a massively parallel computer, why not test all of them? And that's in fact what Hoffman did. There was another factor in Hoffman's decision too which was that, which might convince the rest of you that you might want to test all of them, which is the intended application of this thing include tasks like targeting nuclear missiles.

So let's test all of them. And he did that. It actually only took 6 minutes. It was a fast computer.

And so they wrote an oracle that computed integer square roots a different way – that's what slowed it down to 6 minutes. And they had the built in processors, and they were doing calculations, and doing calculations, they'd chat. And they found two errors. One was in the high 30 thousands and one was in the not quite 4 billions. Two errors that weren't at any boundary, and weren't at any simple bit pattern. And what had happened in both those, is one bit had been miss-set. It was a low order bit, if in your calculations you added this result to any other result, then that bit was rounded out of the calculation and nothing bad happened. On the other hand, if you multiplied before you added, then the incorrect bit propagated up into something that could be seen as significant. And in two places, the error survived all the rounding errors thereafter. Neither of them near any boundary case. The only way they could have found those was the way they did, with exhaustive testing. Now . . .

Fully repeatable. It was traced to a clear error in the code. No question. They miss-set a bit. This wasn't a random error. They had an actual algorithmic mistake. Oops, we blew it. We should have added one bit here instead of subtracting it. Um, now that was great for 32-bit integer square roots. But imagine the problem they had when they had to go to 64-bits. Which they also had built in square roots because now they had 4 billion times six minutes worth of testing to do. Not counting the extra time for doing double precision arithmetic, let's pretend that didn't take any extra time. To do exhaustive testing would take 4 billion times six minutes – sorry that's not going to work.

So they had to sample. They couldn't do exhaustive testing. And so when they went into their very massive random sample, with a certain amount of stratification, everyone working on that team realized, they may be missing of these special case problems. The risk is real. It's more likely that you will find errors at boundaries, but everybody who has done extensive random testing of values and variables, reports that there are sometimes intermediate values that wouldn't be covered by boundary tests that show up as bugs for some cases special case reasons, and others algorithmic reasons but that only show up in the mid-range.

So it's not a silver bullet. It's just a strategy that allows us to find most of the bugs with not very many test cases. Imperfect, but very useful.

Skills involved – Lead in to exercises

If only equivalence class analysis were so simple. There are several different skills involved in equivalence class analysis. And all of them seem to be pretty challenging for at least some people to get. When I say at least some people, the joy of being a professor for the last two years is that I can give students exams and find out what they actually know.

And so I can talk to folks and say, oh yeah they got it. I can stand in front of the room and people's heads go {bob}, they smile, they look alert, they' re not sleeping. I say do you understand, everybody says yes, and I give them a test and they're clueless.

So, I'm still learning what the components are that people have to figure out to be able to able to do successful equivalence class analysis. Certainly one thing is identifying variables that you can then partition. And there are a whole lot of different kinds of variables. We have input variables.

And yet, even though the classic descriptions of equivalence class analysis have been in terms of domain and input variables and their ranges, *the fact is that everyone who is a skilled tester in this line goes quickly into output variables to.* Because we want to test results. The point of the program is not to filter the input. That's the capability programs have to have. But you don't buy a program for the privilege of having it tell you that 32 is a bad date in any day/month/year field. You buy the program in order to have it do something with that date, like print a check on that date. And so you need to look at the outputs if you want to look at the benefits that the program provides.

If you're going to test a program and its variables, you end up needing to think about its input and its outputs, and by the way you need to think about its environment. The printers, the operating systems versions, the network connections, the amount of traffic in a system are all examples of environmental variables. They are not things that are manipulated inside the application, they are givens from the application point of view, but they can affect whether the application works and the application can affect whether they work.

Equivalence classes for configuration testing

So of course you want to test them in conjunction with the application. You can have enormous numbers even within one variable, what printer are you testing with, you can have an enormous amount of tests just on one. Think of all of the different patch levels, across all the different Microsoft operating systems that application might test and, or if we decide to think about the other operating systems, how many versions of UNIX are there are? For a while it was beginning to sound like it was an uncountable infinity.

How many different mainframes are there in terms of the characteristics of the mainframes? You end up with people customizing the software on the mainframes so heavily that mainframes are like the low riders of the computer industry. Developing a program that's going to run on mainframes is developing something that is going to run on a whole series of wildly different customized machines. Those are you're environmental variables and you have to accommodate them. Those are very complex multidimensional environments and if you can't map to yourself what kind of equivalences there might be you're going to find a huge irreproducibility problem in the field, as the program runs into problems that you simply did not analyze.

So we have input variables, output variables, environmental variables, and then we have pseudo-system-level variables like the file system, the functionality that was technically hidden from you that is important anyway. How much memory is there? How much are we using? How much is left? When you ask the "how much" questions you have an equivalence class analysis that you can do. But the "how much" answer is not part of the obvious externally visible thinking associated with any program. How much space is there on your hard disk? How does your hard disk behave when it's almost full? How does your system behave if it fails to write something you needed to write to you're hard disk? Those are examples of system level variables, they're not environmental, they're operating system management variables of how your program interacts with the core part of the system. And different applications will interact with the operating system to different degrees of success and different levels of fragility. How fragile is the application with respect to the operating system and how fragile is the operating system with respect to applications of this type? All those are subject to equivalence class testing.

So the first set of challenges is to figure out what all the variables are and I can assure you that even the most experienced equivalence class testers that I know find this enormously challenging and have very little confidence that they've come up with the list, the full list, when they analyze any program.

You’re always dealing with a subset. Coming up with that subset, here take the program, start brainstorming all different kinds of variables, input variables, output variables, and so forth, is an interesting task in itself. You can do drill based exercises. I think of these as homework, not as stuff I would do in class. But you can certainly use them as quick, first-thing-in-the-morning, do-you-remember-anything-from-yesterday-type exercises. Let me tell you a field that has its values between 75 at 200 -- what’s the lower bound, what’s the upper bound, what's the one just below the lower bound, what's the one just above the upper bound?

Timeout example

Now let me take you to a numerical field where you're not doing the input, but it still numeric. We have a process that times out. I'd like to sit at this machine and tell me what the boundaries are if we know there's a 10-minute timeout period on input. And people will say I think I should type a key and wait 9.999 minutes, and 10 minutes, and well before 10 minutes, and well after 10 minutes. You say great, let's try a test for that, what would you do?

Well the interesting test for that kind of thing is to send some message to the process that's waiting for input that would be handled differently if the process is actually waiting for input or if the process rejected the input.

Here's an example of a process that would handle differently. The telephone. Pickup and start dialing. Your telephone is in a state where it’s either listening for the next key or it's not and you can get calls waiting while you’re dialing, you can get calls waiting while you're talking, you may get something else if your telephone is that reorder or in some state where the computer on the other side is saying "Cannot complete as dialed; please try again."

Different phone systems have different timeout periods; they have different timeout rules.

The simplest case is the 60 seconds timeout rule. This system starts timing from the minute you punch the first key and 60 seconds after you finish hitting the first key and you haven't hit a valid phone number seven digits, eight digits, 10 digits, or whatever the valid recognizable number is, it will either dial the digits you tried or it will give you tone saying this doesn't look like a phone number, hang up and try again.

So here you are with a 60 seconds range. Set your system up so that at 59.999 seconds that last tone comes in. Does the last tone come in just after the phone is actually typed up, just before the phone is typed up, or just as the phone is tearing down the session but hasn't quite gotten it done. At that edge you might find a value in and of itself. Now throw in an interrupt, a phone call from another device and see if the timing on this phone changes just a little bit to see if it allows part of the system to stay alert even though the rest of the system has said, it is after 60 seconds -- we're gone to launch. If part of the system thinks it is still listening for digits and part of this system thinks it is not still listening for digits then you have an opportunity to crash that part of the system.

If you're in a company where they do real-time systems, having people talk about those, and have them analyze them in terms about smallest time, longest time, what happens if boundary cases in time can be interesting. If you do them like I just did, it's not an exercise. If you try to do this with a group of people who have never tested real-time systems, it's a mystery; it's not an exercise. If you do it with a group of people have real-time systems, they go, "Oh, yeah, gee you could think about that as a domain, right, I'd map it out this way." That's the result that you want to have, starting to draw sketches. They have to be halfway there or they won't get there.

Background for exercises

So simple analysis of variables is the first, easiest, most direct and most boring of exercises. Good for homework.

The next piece, which I give as an assignment, is like simple analysis. I ask for just a little bit more. Here we are with a function of 20 to 50 and I say OK what I want you to do is write up a boundary analysis chart and give me the chart. And the chart is going to be a document that you can hand to another tester because the wants to test the 20 to 50 deal. Think of it once a get somebody off to test it.

So in your test documentation how would you create a chart for somebody else to use? And the classic version of that chart was published by Glenford Myers in his book *The Art Of Software Testing*, where many of the best ideas in the field come from even though it was 1979. But here we have a variable; we have what Myers calls the valid class and what I call the main class or what in a use class would be called the happy class or happy case, 20 to 50 in this case. And then we have what Myers calls the invalid class but sometimes it's not a matter of valid vs. invalid, it's a matter that gets treated differently. And so that's more effectively talked about as alternate. And then you always have a call for boundaries so what might happen here are would be 20; the alternate classes would be less than 20, greater than 50, and not a number.

We can go through the same list that we did before but I'd like to not bother, but we have boundaries like 20 and 21, 50 and 51, 19 / ; and then we have notes. And what I ask people to put in the notes is a statement, why do you think this is the best example of a test for this range? Now if you're doing a simple linear field, you would think the answer to that -- why is this the best representative -- would be an obvious one, it's a boundary, there's an extra reason why you're there.

Surprisingly, it takes people one or two practice sessions to even to get that in a way that they can say it consistently. But as soon as you get outside of 20 to 50 and then talking about 19, 20, 50, and 51 as soon as they get into things that are not immediately on the number line, what's the best non-number? Well you can talk about the best non-number in terms of the ASCII code and some folks might especially since we have covered that already. Some people might talk about the best non-number in terms of "A" is the most similar letter to a number because it's one of the letters closest to the numbers in the ASCII code.

Whatever they come up with as the reason, what you ask them to do is for any test case they propose, *have them tell you what class is that test case a member of, and why is it a best member of that class.* That turns into a hard question for people to answer. In my experience with graduate-level students who have bachelor's degrees in computer science, we're not talking about people who have never had any thinking of testing, we're not thinking about people who could never code their way out of a loop, we're talking about people who came in with good degrees from reputable universities who can code quite well, thank you very much. Many of them with industry experience who for a simple one like this, still cannot tell you on an consistent basis, what member of a class or what category they think something belongs to and why it is recommended as the best.

The simple one gets the juices flowing. You start them out and give them 20 minutes to try it, take it up, have people cross look at each other's papers, walk around a look at a few yourself, point out that there's not an explanation for some of it. Describe what some common problems, are again I really hate debriefing were students stand up and then read their own stuff or read their neighbor’s stuff, what do you have to say, what do you have to say, what do you have to say. In three a student class debriefing works great. In an 8-student class if you have more than 4 people debriefing, it’s just boring. But if you come up and say, here are three examples of problems people had in the random, let me give you an answer. And post a solution by one of the students but I am not going to tell you who that might embarrass them, then critique it and folks can learn from that. Then you try to find an illustration from every person across different exercises so that every person sees something of their own work and gets it. Then you get to comeback with a slightly more complex variable and try it again.

Slide 5.21: Optional Exercise 5.3: Myers' Triangle Exercise

During the break I was asked about another example of a domain class exercise. It is one of the classic exercises in the field. It comes from Glenn Myers book, the Art of Software Testing, and it's called the triangle problem. A triangle problem seems like a very simple problem. We have a program that will accept three numbers, call them A, B, and C. You're going to enter values into A, B, and C. We will pretend for the moment that all you can put in are integers. It will dutifully reject anything that is not an integer and the program will feedback to you whether the triangle that you entered is an isosceles, scallion, or equilateral. Of course if you use this exercise, you'll find for your students that isosceles means a triangle that has two sides the same. And equilateral means a triangle that has three sides the same. Scallion means a triangle that has no sides the same. Anyway, the task for the students is to list a sufficient set of tests to test this program and to find out which parts of the program are isosceles, scallion, and equilateral. Of course, the output we have forgotten to mention is not a triangle. But you're bright students will figure that out. There are a lot of "not a triangles" that they can come up with. One of the most common "not a triangles" that I get from students is 1, 1, 3, which they call isosceles. Now an isosceles triangle whose sides are 1, 1, 3, looks like this. It's got 3 on the base and 1 and 1 and they just don't match the sides are not long enough. The sum of these two sides has to be at least as big as the base. If it was one and a half and one and a half by three, they would match but they would match by being a line; they fall directly online.

So, when students don't get that one, I smile and say this shows why you need domain expertise. This happens commonly in specifications. The piece the programmer had in mind is something she was controlling for deliberately, like the case statement: is it isosceles, is it equilateral, is it scallion, she'll specify that. But the piece that she assumes you know, everybody understands geometry knows that the smallest two sides have to be at least as big in their sum as the larger one. That's just domain knowledge, everybody knows that. We don't have to put that in the spec. But as soon as you come against that spec and you're not knowledgeable in that domain, you'll come up with stupid test cases. And of course if the programmer didn't know it either then you'll come up with troublesome test cases.

So the first learning I get for my students is that many of them don't have the domain knowledge and it gives them a chance to realize that. My structure by the way in my academic class, which is the only place where I use Myers program, is to start people out in the first 10 minutes of the first day with a Myers program. And I say this is your opening quiz, after all this is a testing class, I start with a test ha ha ha. And they go, darn professor. And then they write-down the test cases they would use and the other thing I asked them to write is what percentage of the task they think they should run they've covered. One of the amusing correlations is the people who give me numbers like 100% coverage typically give me three or four cases at most. The people who know the least think that they have tested the most thoroughly. Many of the students have bachelor's degrees in computer science and have written programs much more complex than this themselves. They come up with one case for isosceles, one case for scallion, and one case for equilateral. The smartest of the students also comes up with one case that's not a triangle. And then they say that's enough here it's 100 percent. Whereas students who give me 25 end up saying, I don't know what the percentage is but I'm not there yet. They might not express in that exactly those words, I can't give you a percentage, or they might say I guess this might be 50%. They're a lot more tests but I'm out of time. And then I basically tell them it's about 10 minutes and I stop when I see that are people are looking at me more than they're looking at their papers.

I take their brainstorms home and bring them back the next day. And then we do a takeoff and class but before we do the takeoff and class, we've gone through an exercise like the 20-50, we've developed the test ideas matrix, and said then I come back and say OK, how would we come up with a test ideas list for a triangle? And we do a little bit more in class and I point out some of the cases. The reason I am mentioning this on the tape is that Glenn Myers in his book gives you 11 classes of test cases that he thinks collectively cover interesting tests for these. One of the pieces of an assignment I get my students is what really interesting cases did Myers miss? And most of them can find at least one but if you go through his notes you can probably find at least one if you can't don't worry, just don't asked them to tell you which ones were missed. Just don't ask them to tell you which ones were missed because then they will say what do you think it was and you'll be lost. And after the first time you teach this, somebody will come up and say, "Well teacher, isn't this an example of something, shouldn't it be on the list?" And then you have one.

So the next time you can ask for it. Um, there's a risk of spending too much time on this exercise. I'm told that there's a consensus within teachers who teach out of the British Computer Society to stop using the Myers example because they think it teaches people the wrong thing, especially in the hands off inexperienced instructors. The risky thing that it teaches in the hands of inexperienced instructors is that you should test all 11 or 28 or whatever the number is that you come up with all of the interesting cases and they're a lot.

Slide 5.22: Exercise 5.3: Myers' Answers

Several classes of issues were missed by most students. For example:

- Few students checked whether they were producing valid triangles. (1,2,3) and (1,2,4) cannot be the lengths of any triangle.
- Knowledge of the subject matter of the program under test will enable you to create test cases that are not directly suggested by the specification. If you lack that knowledge, you will miss key tests. (This knowledge is sometimes called "domain knowledge", not to be confused with "domain testing.")
- Few students checked non-numeric values, bad delimiters, or non-integers.
- The only boundaries tested were at MaxInt or 0.

For example this side could be zero, this side could be zero, this side could be zero, any combination of that can be zero. This could be bigger than these two, this could be bigger than these two, this could be bigger than these two.

This could be maxint although actually Myers doesn't get into maxint. the whole notion of buffer overflows and that kind of stuff is not there, by the way that is one that is not the list now you don't even have to wait for your students anymore to talk about buffer overflow problems.

But the sum of them can also exceed maxint, why does that matter? Because even though all of them individually or find, there's an internal calculation. Let's suppose that we try to have an equilateral triangle that is max-in, max-in, max-in. What's going to happen what we add this side to this side to see if it's his biggest this side -- can't do. How does the deal that? It's got to be special code.

So, it has to behave at the little different than just a simple addition of two sides. And it might reject it, or it might have some error handling and do the analysis in a different way. It doesn't matter, it is a qualitatively different test case from two and three and four.

So, we can come up with lots and lots and lots of examples. If the message the students get across is there are umpteen-gazillion tests they're at risk. There is the same risk that we talked about earlier with a test ideas list if you try to test them all, you lose your ability to do any other tests in the program. And what you've created is the excuse for never getting done. Some testers live by the excuse of never getting anything done. They walk in and say, "My project is a failure from the start. There are all these tests. I have to do them all. Those evil project managers won't let me do all the testing I need to do. They're so bad, I'm so sad. I better put down lots of paperwork to explain why I can't get things done and tell them that I need more staff and of course, do the best I can and test it." This should not be a course that trains people to be victims. They have to come up with a sampling strategy. The way that I use the Myers exercise, is that it hits in my sequence of exercises I start them out thinking about the question, how do we come up with out of a big population of tests of something that gives us some percentage of coverage? And then we look at how we guesstimate coverage out of that. We also look at how we come up with a long list of tests. But then we come back to the question of representatives. People want to say let's get big numbers, great, there's a population of possible test that are big numbers size. Let's pick one and make that the best representative or at least the representative we are going to use.

So I tend to use this as an exercise of test case reduction; whereas many of the folks who teach from Myers make it case of test case expansion. Myers uses it that way too. In the opening exercise he says come up with a list. And then he says, see, you didn't get them all. Most experienced testers don't get them all, you're not thinking clearly enough. Well that's a useful thing on the first half day of lecture and then we have to get them sophisticated enough to know that even if they get them all, they cannot use them all.

And so now we have to go to the reduction part. And my sequence the reduction part is another case of: give me the test, tell me the category of additional representatives, tell me how big the category is, and tell me why this is as good a representative as any other, then give me at least one example of something you think this is a member of the class too but this one is a better example of a member from that class.

So in the equilateral triangle instance, it's interesting to look at a triangle who sides when you do real numbers look like 0.0001, 0.0001, 0.0001 to see if they can handle little, itsy-bitsy sides. Another example of sides better identical, max-in, max-in, max-in. You know, we're looking at extreme cases were the curve is getting tough.

Those two cases might be the nastiest examples of the general class of equilateral triangle. Maybe you can come up with another nasty example, that's great.

The students come up with a variety of them and I say pick one and tell me why you think that's a particularly good representative. And the goal is to come with 10 or 12 tests that will become the representative group. We have 15 minutes to test the triangle and then we have to get on with dealing with the square. And how many geometric shapes are there in the world, we are going to be testing each one. We'll put the program in the little perspective -- this is the program that can generate every geometric shape you can ever imagine.

How many sides can a geometric figure have? I think it's infinity. So take a little time for each one you are going to run out of time before you run out of shapes. Let's start with a triangle. And that gives them a sense of the urgency. And some people say that's not realistic, so I say OK let's use a word processor and let's think of every document you could ever print and they go, "Oh."

What's the representative set of documents? -- it is the same problem. How much time do you want to spend on the document that has three words. There are a lot of tests for the document that has three words. How many letters have you written that has three words and that was all? That's what you're testing with a triangle program. It's good to know it can handle three words, but at a certain point you have to move on. Give me 10 or 12 test that you think are better given that we have to move forward and tell me for each one why that's a good one.

That's my use of that exercise. As with some of the other exercises, the grading is a critical part. It's not just writing it down and it's not just taking it off at the front of the class. It's looking at some of the very strange answers that people give and trying to figure out how they could think that this is representative of that class much less a good representative. And then twist your head a little for you to see how this one fits and say back to the student, that's an interesting theory but maybe you could think of it a little bit differently.

To some degree you can achieve that by having people trade notes and rate each other's stuff. To a large degree if you're doing exercises with that goal, you're going to have people turn stuff into you and in your copious spare time at the hotel, you're going to be making notes on exercises that they can look at the next day. Now as a professor I can be that happily. My students expect that. As an industrial instructor I've never even tried it. I don't know how people would react. But that's the way to give them the best feedback. If I was really focused on equivalence class testing, I'd probably give them one example where I could actually see what their four or five classes were and make notes on them.

Slide 5.23: Optional Exercise 5.4: Equivalence Analysis with Output

Here is an example of a more complex situation.

Let's divide a variable "K", that is an integer variable, however many bytes an integer is in your system doesn't matter, it's that many but it's a result. K is a product of I and J. I is also an integer variable. J is also an integer variable. K is an output variable; you can't enter anything directly into K.

For an equivalence class analysis on K, you have to think what you would enter into I and what you would enter into J to drive K through the interesting values. This is a classic problem that we face all the time and testing. We see a report and we say how can I get these three values into this report? And the only way you can do that is to figure out what inputs you need to get to get these three values. It's a normal thing but what they're going to have to do is to start out by saying OK, K can run from minint to maxint. If you don't want to play with negatives, no problem, deal with K as 0 to maxint. (By the way, I can also run from minint to maxint and so can J.)

So we have K, who's values can be minint to maxint running through zero. And we have other interesting values like 1 and minus 1. Now how many possible values are there of the pair I and J that can give us a result value of 0. What's the set?

Anything where I is 0; anything where J is 0. All of the I's and J's where I draw them give us 0. It's a huge set.

What's the set a gives us -1 as the answer?

-1, 1 and 1, -1.

So some of these sets are big and some of these sets are small. And there are some pretty interesting invalid ones. What is the system going to do when you try to multiply maxint times maxint? You can't have a value K for that, right it's not to work. In fact you have an interesting question, how do we get to the biggest possible value for K?

Let's think of a small maxint. Let's suppose that our integer was only one byte. The largest possible positive integer if we have signed bytes is 127. If that's too big for your students, think about something that ranges up to 7. It's just as good an example except we are fitting into four bits I think positives and negatives, maybe six bits or something I forget. How do you get to exactly the maximum values that K can have, well maxint times 1 and 1 times maxint will do it.

But as soon as you start thinking about multiplying at a big value of I, or mid-level value of I times a mid-level value of J, you discover that people want to multiply the square root of I or the square root of maxint times the square root of J, and that can't work because it's an integer system. The square root of 7 is 2; it's 2 point something. But in integer, poof, the point something goes away. And so the square root of 7 times the square root of 7 is 4. Well, that's not maxint.

So we end up saying what's the largest integer and started playing around with square roots and that can get into fun.

What people end up saying is that for the results they end up with sets. And some of those sets have lots of numbers and some of those sets have very few numbers. *And now that you have a set, what's the best representative of the set?* Well, for the 0 case, I don't think there is the best representative for the set, I think they're all about the same so pick one. Maybe the boundary is (0,0); maybe the boundary is (maxint, 0) or (0, maxint); you can play with those, that would be fun. But I'm not sure that any those are more likely to fail than any of the others, so I might randomly sample from that group. But what you can see is there is a group that you've got to sample from, and if there is another group you've got to sample from it, and if there is another group you've got sample from it, in each case it you have a set. It's not exactly clear what the boundary is. If there is the boundary it's a two-dimensional boundary. You have a set you have too many across all the sets to be worth testing.

And so you say OK I have a small number of sets I want to pick one for each. And then I want to justify why (1,1) is the interesting case to use if that's the one I pick. Why did I pick this instead of the (-1,-1)? And maybe my answer is I'm going to pick (1,1) here and (-1,-1) here. I'll cover my negatives in some other place, I can only afford to do few.

So have people justify that. Again, have them do that in a group. This turns out to be an extremely challenging case, the first time they try it. People will get frustrated. I take my students out to a café, I take them away from the lab atmosphere. We go to the Sun Shop Café, we start a Saturday mornings and I pay for their breakfast and while they work on this thing. They stay till noon or 1:00; we have flip charts that we set up. The owner of the cafe thinks it's kind of cool that we have all the stuff and we can take the back part set of tables to play with. And I sit in another part of the cafe and when they get really frustrated, they come by and talk but most of the time they talk to each other.

And so I've heard from across the cafe some loud discussions about how to handle this problem the first time. If you really want your students to master a problem like this, then let them fumble with it, pick up what they did well on this problem and then solve it for them.

Now let's do

$$K = I / J$$

and let them do it again. They tear their hair out more but they're likely to get a lot further on this one. And if they don't do this one, I know it didn't stick. Because when I just give them this one and then I have a final exam or have another problem that's another variation on the theme, they don't necessarily do well. When I give them this as an assignment and this as a follow-up assignment, they come to the exam with skills then.

It takes that much. This is a lot of work to go through in a class to get to them finally to the point where they can look at a report and say, how I test this is with these classes. Now is that an important skill? Yes. But were not talking about an hour's work of practice. If we added up all the time they spend on the exercises that they do to the place where they get to this, they probably spend a day, including feedback time. How much of that day you want to put this class through, that's up to you.

But any subset of this is going to be valuable in getting the growth in skill and without any of it they might articulate, they might say, oh yes I understand equivalence class analysis but that doesn't mean that they know even where to start. And anything less obvious than 20 to 50 integer range. And here's another interesting question to ask them, this same group. I'm going to give you any input range that supposed to range from 20 to 50 but we’re going to structure the field in a way that you can type any amount of characters you want and press return and that will give you a message if it’s out of bounds. What testing do you do in this case?

What you have is two boundary cases here. You have 20 to 50, that's the boring obvious one. And then you have the buffer; the field that you get to put the data into before it gets processed. And that one obviously has a wider limit to it. Whenever you see two fields disguised as one, you see an opportunity for all kinds of fun testing to go on an all sorts of serious errors. Now you start asking the question, what is the biggest number I can put into this, and how will it respond?

If I give students the task, find me the upper limit and then find me a value that exceeds the upper limit and see what it does. People don't necessarily have a good strategy for this. But here's a bad strategy. 1 that's OK, 2 no that's OK, 3 no that's OK, 33 yeah that's OK, 34 OK. Here's a little better strategy but it's not very timely 2, 4, 8, 16, eventually you'll get to something pretty big but it takes a while.

I end up with a strategy that goes like, it goes from 2 to 50, OK, let's try three digits. Oh, it takes three digits. Wow. Let's try 6 digits; it takes six digits, terrific. Let's try 12 digits, gee it takes 12 digits. Let's try 24 digits. Hmmmmmm. Now up to this point I am cutting and pasting but pretty soon when I get into more than a few digits, I erase that and go 123456789A. If I possibly can, I will put in larger. It will be rejected anyway. And then it's 123456789B so I like to see what actually took when I scroll through. In a little bitty scroll box with a number this big, you can't tell unless you have identifying marks. But now my pasting is going to be paste nine digits type a letter, paste nine digits type a letter, paste nine digits type a letter, paste nine digits type a letter and on we go. And just keep pasting in large lots. Maybe I'll end of pasting 100, lots of 100 inputs. Eventually I'm going to get something that is too large for it to process. But the first too large for the process that's just fine for the starting point. It still let me put it in.

So maybe it's 255 characters it tells me I shouldn't have given it that many characters. Which means 256 is a great case to see what it does. And indeed in some fields that I've worked with, it gives you different messages until finally you crash the operating system, or you crash something else. The process of an organized and determined search for boundaries is an important skill for anybody that has ever had a program that didn't specify the boundaries for the fields. Which probably is everybody with probably every specification they’ve ever worked with, some field is unspecified.

So you end up saying what is the down, how you handle things that are obviously out of bounds, and is this boundary a reasonable thing.

So giving people search tasks is practicing another skill they have to use on the job all the time for boundary cases. But it's going to take time as an exercise. That one might well work as an interesting exercise where you have the field, and you have people tell you what the next thing is to type in. at certain points you comment, this is getting really tedious, can’t we come up with a strategy to get me to the next level of interestingness as far as bigness a little more quickly. The -- get me to the next level a little more quickly -- is a fundamental issue of domain testing, equivalence class testing, and with everything else. But the point for equivalence class testing is to get ourselves into the next group as fast as we can because we have way too many possible tests so we're trying to go from group to group quickly. The more time we spend with one group, the less benefit we're getting out of the strategy.

So that's another class of exercises we can try to come up with. Certainly on an exercise basis, it's an interesting thing to do. In person-to-person coaching, I've done a lot with happy success. With groups it depends on people really being willing to walk with you and keep the focus. I don't think this will work in a group of 20. You will have three people participating, and 17 people wondering what their e-mail is like, whether they can get out of here and make connections on their machines here.

So there are a bunch of other examples of tasks that people perform when they are do equivalence class testing that are in the student notes and you can make exercises out of any of those. If you don't give people some of these exercises they will not master the skills. And if you give them some of these exercises and you give them enough time to get the full value then you're going to take a few hours out of your day and you will lose the pace of the here's a technique, here's a technique, here's a technique, here's a technique, see that was an interesting tour. They see a budget hills instead of the mountain range.

So that's at a trade-off different instructors will make differently. When I taught this kind of material in-house, my general approach has been to go a day early to the company site, sit with the sponsoring manager or managers of the class. Walk through the course notes with them and ask them questions like, how important is this skill and how much time can we spend on this skill? If we did this with it, will this benefit you are not? Do they need overview more or do they need drill more? What do you hope they will come up with at the end of four days and focus on that material. You can't practice everything. Any comments are questions?

Slide 5.25-6: Test Techniques—Specification-Based Testing

You can't do spec-based testing if you don't have a spec. Therefore not every test group will do spec-based testing. If you have a spec, and that it's close to complete, and it's up-to-date, and it's intended to be accurate, then there is some value in making sure that the program conforms to this spec.

You might remember the last time you ordered books from Amazon.com. On the outside of the box was the invoice and it said there are three books here and these are the titles. What would have happened if you opened the box and saw four books and they weren't any of the three that you ordered and they were more expensive? Would you say oh good I got a great deal? Or would say what they tell me I was getting these books and I got those. In fact, what if you ordered 10 books and they give you a partial shipment? They said we will send the other seven next week but here are these three. You opened the box and you have three books from your order but they are different from what were listed. Would that make you nervous? Of course it would.

The nervousness comes from the mismatch between the statement of what you are getting what was actually delivered. Anytime we tell the customer what they're getting and it is not what they get, it's a bad thing. Let me quantify a bad thing in some other ways. If you sign a contract with the customer and you promised to deliver 1000 features these 1000 features, and then you don't, that's called breach of contract. If you shipped the product to the customer without having a little checklist of the 1000 features checking to make sure that each one is in the box and works at least through the happy path. If you don't do that much, you're begging for breach of contract lawsuit. "I will ship you $1000 of stuff. Here is $800. Your order is complete." Would you accept that as the customer? It's the same thing. Now in mass-market software, we have another interesting piece, because in many states the user manual is taken as the binding specification. For other products, that is certainly true.

The classic legal case happened in West Virginia somebody went into a diamond store about a diamond ring. And the interaction between the jeweler and customer went like this. "I want to buy a diamond ring for my anniversary. I want it to be really pretty." "Good" the jeweler said, "we have a big stone right here and it's really pretty." The customer said, "That is a big stone and it is very sparkly. I like it. How much as it?" They agree on a price and they wrap it up and the jeweler wrote an appraisal, which the jeweler put it into the box without even telling the customer about it. And in the appraisal it said the diamond was of the VBS quality. The VBS is diamond talk for really good.

The customer took away, didn't know anything about the statement. Four months later he gave it to his wife. She opens the box, likes the big rock. The little piece of paper falls out of the box. She sees the little piece of paper, folds it back up and sets it back in the box and ignores it. A little more than three years later her daughter is trying to get the diamond appraised. Probably the mother died and they were now doing appraisal for that reason. She takes the diamond to the jewelers and the jeweler said, "It's a nice diamond but it's not VBS quality." The daughter says, "We were sold a diamond that was VBS quality. This was not the VBS quality." She goes back to the original jeweler and says, "I want you to replace this rock with a VBS rock." And the jeweler says, "Well, it's not really a VBS rock and I'm not going to give you a new one because that would be way more expensive. I tell you what, I'll buy back the ring."

The price had gone up at this point. This was a good thing for the jeweler; well it might have been a good thing for the jeweler. But the daughter said, "You wrote down VBS quality. That's a description of the product. It's a spec. I am entitled to the product specified." "I remember that sale. We negotiated this one. It was right here in the case and you got this one. It's a nice diamond."

So it went to the Supreme Court of West Virginia and what the Supreme Court said to the jeweler was, "Sorry. You wrote down this is VBS quality, even if the customer didn't see that before they bought the product. But the product was delivered with a description of the product; and it's binding. Of course it's binding." Well what's the difference between that description and all the claims we making user manuals? I don't think there are any.

You might say: "In the diamond case we have an accepted standard whereas in the user manual we do not. While the standard the judge applied to this was a very simple standard, is the statement true or false." Legally, the definition of an express warranty is the statement of fact that applies to the product as made by the seller to the customer and becomes part of the basis of the bargain. The meaning of the statement of fact is you can prove true or false, you can prove true or false in the case of the diamond by going to one or two other jewelers and saying, "VBS?" And they go, "Well plus or minus one grade, yes." For a feature that crashes, you can go out to anybody who uses the software and say crashes and they say, "Yes." And you look at the manual and it says, “Do this and it will do that,” but instead the program crashes. That's an easy standard. If the manual makes the statement that you can prove true or false and the manual statement is false, then you have what you need. The manual creates a reasonable expectation about the product on the part of the customer. The customer will be cranky whether or not you can enforce it in court.

(OTOH, if the manual has an implied use then there is no statement of fact. If the manual says you're going to have to stand on your head and twirl four times before you can do it but it says stand on your head and twirl, you say this is ridiculous, then the manual is saying nothing except what's so. You might not like what's so. But if the manual says it accurately there's no issue of this kind.)

Where there is a clear cut statement of fact -- something that a reasonable person would read and say I know that this is making this assertion. I can imagine the test that would tell me whether this assertion is true or false. There is great wisdom in checking your software to see that it's true and not false.

How does the tester determine what the claims are that we made. Well that's one of the skills. Just like one of the skills for function testing is figure out everything that you can test stand alone. One of the skills for verifying every claim is figuring out what the claims are. Another skill is, figure out with the claims mean. Ambiguity analysis. But given that you find a claim, your task under spec driven testing is to check.

I want to get back on the manual in a different way. Bill Rose, the president of the Software Support and Professionals Association, wrote a book in 1990 on tech support. And the last time I talked with him about this was in 1998, moderately recently, about what things drive tech support representatives crazy. And he said that bad user documentation is this single biggest cause of difficulties between customers and tech support representatives.

When I was writing a book on software tech support and software consumer protection called *Bad Software*, I interviewed several tech support managers from several companies. I heard with this constant story time and time again, actually unexpected to me at the time. The biggest reason why tech support folks quit: *customers see something in the manual that's false and it really false, the writer screwed up it’s not the software.* The customer now expects the software to behave in this way; it's not even a good way. The customer called up it does and says, “The software doesn't work.” The software tech support says, “Yes it does; the manual is wrong.” The customer says, “Oh, yeah right, you just say that -- I trust *you*.” And customer goes through this very sarcastic conversation with the tech support rep, because in black and white in print it says that the program works this way when in fact the program works this other way. And even if this other way is a perfectly good way, the customer is dissatisfied because they were told inaccurate stuff.

That interaction repeated many times during a day gets the tech support people really frustrated. People scream at them, people swear at them, people call them liars constantly and you end up with folks who say I can go to another company. I don't have to deal with this abuse. I'll go someplace where they actually check their manual. Stunningly, at the Customer Care Institute they studied mass-market software publishers and over a three-year period. They found that between 44 and 54 percent of the software publishers they studied, 300-400 publishers at a whack whack, basically have the publishers never submitted their documentation to testers.

The mass-market provides my best argument if I'm going to take a company to court for breach of contract and the number one leading cause of software tech support anger/frustration and people quitting and so forth and we’ll just say, “Oh, let’s not do that.”

Let me just add one-piece, my experience with consumer software is it takes about 15 minutes per page to do the verification against the manual for one pass of the manual. It's relatively cheap and yet half the industry still doesn't do it. Part of the response to this is now we get skinnier manuals -- less to check. But we also get a faster escalating number of tech support calls because people can’t figure out how use the product. That's a different class of issue.

The issue I want to stress here is, *if you set a customer expectation you need to check the customer expectation.* If the way you set customer expectation with a spec, then the way you check the customer expectation with spec driven testing. If you're setting expectations and not checking them, then you're engaged in high-risk behavior. And close to half of our industry does not bother checking. They don't think they have specs so they don't do spec-based testing. Well, so even companies where they don't write formal specs they have a need to do spec-driven testing, often a big need.

The major things that are missed if your main approach is spec-driven testing are as follows. If I have a claim that is made in a spec and I check that it is accurate, great. I've now tested the equivalent of the happy path.

Have we tested anything else? No. What about all of the behavior of the program that is not specified in the spec that we've left to common sense? If the only things that we’ve checked off are the things related to the spec, no spec can exhaustively describe every aspect of program behavior. Some of it is left to the good sense and reason of the people who are interacting with it or testing it to.

If you are thinking about the implications that can be drawn and we are don't test those, some of those things will turn out to be failure cases. That's problem 1, anything not mentioned in the spec doesn't get covered if all of your testing is spec-driven.

Problem 2 is: Even if it is mentioned in the spec, if you tell me that this will add two numbers, use this function and you can add to numbers, OK great. Do I add 2 plus 2 or do we add maxint and maxint. Either one answers the base question, does it add two numbers. But if I'm not going to do harsh tests, even though I can check off an item to say yes I covered that element of this spec, I'm not covering with enough power to have confidence that it will be met under every circumstance that a reasonable person would use. *And so in the one case spec-driven testing doesn't cover anything missed in the spec and in and the other case spec-driven testing does not necessarily cover the more troublesome cases that weren't separately called out or implied in the spec*.

Slide 5.27: Traceability Tool for Specification-Based Testing

One last piece on spec-driven testing is the notion of the traceability matrix. There are lots of different traceability matrices out there. There is a general concept to the matrix. I say variable 1, variable 2, variable 3, that might be requirements statement 1, requirement statement 2, it might be spec statement 1 or spec statement 2, it might be line 1 in the manual or line 2 in the manual. By what we have all are a bunch of statements that you can test {across the top} and a bunch of tests {down the side}. And we say that it turns out that the first test actually involves these three claims, and the second test involves these two, and the third test involves these three.

And so you show in a matrix what's been tested by what. This is called the traceability matrix. The first benefit of a traceability matrix is that you can trace back from the claims to the test cases. Why is that interesting? Because one day we decide to change the feature that is documented in requirements statement 4 and the first question that comes up in the change control committee is what is that going to do to testing? And you say test 2, 3, 4, and 5 are going to have change. By the way that will cost you $28,000. And they say, “gulp”. If you can trace back, you can understand a lot about side effects, in terms of cost to retest that you can’t understand if you don't have this kind of a tracking of what you're test cases are. Another benefit of a traceability matrix is running on spec items is he can get a sense of this spec items. On this slide I would've had to squeeze too much to get a total line down here but it’s probably the case you have a line that shows totals.

So you say, OK, there were three tests for variable 1, two tests for variable 2, three tests for variable 3, four tests for variable 4 -- this is a pretty balanced matrix. But imagine that we had 1000 claims, 1000 tests, and claim 1 had 800 tests it was involved in, and claim 2 had 1, and claim 3 had 700, and claim 4 had 500 -- that is an under balanced population of test. We have under tested claim 2 compared to everything else we have done; maybe we have a under tested claim 2 maybe we should look and see if we aren't doing enough. Similarly if we have five tests for claim 1, five for claim 2, five for claim 3, a thousand for claim 4, and five for claim 5. Claim 4 better be something like, go to the main file menu and it looks like this that is something you’re going to reach in every test because it's of importance. Or it better be something that is the most critical aspect of the program and if anything is wrong with it somebody might die.

Maybe that justifies testing this thing two orders or three orders of magnitude more than everything else. More often that huge imbalance happens accidentally.

You can look at the profile how much is each statement tested. And say is this tested more because it is greater risk, did we make this imbalance consciously, or we just investing time in this because somebody finds it easy to generate a lot of test cases and they're not thinking about what the balance of coverage is?

A traceability matrix shows you the pattern of your testing, gives you a top-level characterization of every test that you run, allows you to look back from spec items to things that have to be changed if the spec changes, and allows you to compare the thoroughness with which each spec items is tested.

All of those functions have value. It costs to create such matrices. There are of course several tools that will assist in creating matrices like this. Trying to do this by hand without using those tools is an enormous nuisance especially in the face of code that goes through maintenance because the chart has to go through maintenance and that becomes really interesting. You go through all of your tests and all your spec items every time there's a change, so automated support for creating these have a lot of value. The matrix has a lot of value if you are doing spec-based testing. In fact if you are doing spec-based testing and you do not use a chart like this you need to ask yourself why. Having gotten it to the point where you know all of the claims that you're going to test do not tracked them seems very silly.

If the specification includes an example, should we a) test that example and b) test anything outside the example that corresponds to it. Yes of course we should test the example. Speaking as a documentation manager one of the great embarrassments in manuals that we would put out would be the tutorial that we would cobble together as the last thing that we put in the book that the customer would use and discover that the example doesn’t work. I don't think that any group of mine ever shipped a final documentation that way, but we did ship draft documentation to trade shows that way. And it was just really embarrassing. People would walk through the whole thing, it would fail, and they would say thank you and go on to the next booth. The salespeople would say oh thanks a lot don't do this again. Except they didn't say it that politely.

The example of course is just embarrassing if it does not work along with everything else, but the example is probably really easy and if all you do is test the example then you're not going to see the more complex ones that the customer may use. So now we want to test beyond that. It may be that you can check off all the features that are listed in this book without going beyond the examples that are listed, but if you check them all off with those easy cases, there can be tons and tons of errors in the crowd that you would have missed.

We have an interesting trade-off, so how much testing off the bare spec do we want to do? This is the question whose answer depends. For example, you write a contract with someone to do custom engineering. We create an acceptance test. The acceptance test is directly derived from the specification. The customer reviews the acceptance test. There is a fee for the creation of the acceptance test and the execution of the acceptance test before it goes to the customer. There's also a fee for testing in general but as you look at the fee, you realize that almost all of what is budgeted will be spent going through three iterations of the acceptance test over the last three builds of the software.

Should you test beyond that level of testing? Well, if your salespeople have sat with the customer and they say, "If you're really only going to do this level of testing on these features and that's it, here are the risks." The customer said, "I understand the risk, see I signed it, I understand the risk, I'm only willing to pay this much for engineering. If there are few bugs, we'll deal with that later under separate contract. We'll call that a maintenance contract." And if you're billing your customer on time plus materials for your work, then every minute that you spend on testing beyond those thousand cases, is work they didn't order and told you not to do. Either you have to eat that time or they have to pay for something they didn't order which is overbilling.

In that world where everybody has their eyes open, the ethical thing to do is to tell the customer with the risks are associated with that level of testing and then tell the customer that if that is what they want, that's fine. There are some ethical questions if the kinds of bugs you might miss are the kind that might kill people. But if what we're talking about is the customer might lose a lot of the customer's money if it does not work the way the customer needs it to work, the customer says I'd like to save $5000 in order to put one million of my dollars at risk, well they can go to Las Vegas too that's legal. People are allowed to make dumb business decisions. As long as it clear between you what the risk is, if they want to take that risk and their eyes are wide-open and you are happy to take their contract for every other reason, that should not be a variable. You should do exactly the level of testing that you specified and that's it.

On other hand, if you don't have this clear expectation and the customer is basically relying on you to deliver something that works, then merely going through 1000 features at the checklist level doesn't give you much assurance to spec. The have to gauge, circumstances will differ. Why are you doing spec-driven testing is giving you a gauge for how much of this you have to do.

How much testing beyond the spec should you do? One student answers: *I'd like to answer that. I've never worked on a product of commercial quality where less than 80% of the code has been dealing with things like error handling and configurations that were intentionally not documented and very rarely specified. By the same token, all of those things are important to test. And the decision not to document what happens if the network gets flaky or whatever is absolutely the right decision. In other words its something, it's an error condition you want to handle, it's a quirk of installing on Windows 95 that you want to handle and it's something that needs to be tested and there is absolutely no reason in the world to be documented to the customer or any outside stakeholder. I have also never seen that stuff. I have never personally worked on a project where that stuff has been particularly documented beyond saying we want to deal with Windows 95 or we want to deal with network errors or other types of flaky environmental errors.*

So I agree at that level Windows 95 is a supported configurations needs to be documented as the configuration requirements. Personally I know that certain domains, you know, safety critical software will have your people go will be very tight about constraints of that. But in most cases I think they are and frankly I think it is not guidance that we want to get up to say the solution to those problems is to go through requirements documentation for all of those particular cases but it absolutely is good practice to say test them.

Some companies like to have long and thorough requirements documents. Some companies are tempted to have long and thorough requirements documents and found the expense of the development is so high that they now have half page requirements documents because they're not willing to maintain a larger set.

We have a whole population of development methods that we call the agile methods, which are dedicated to lightweight processes and minimizing paper. Carrying the experience from the test group back into the code is something they would do; carrying the experience of the testing group back into documents that describe the product is something that they would only do if there is a clear stakeholder benefit to justify the time to the extent that there is no more valuable use of that time on other things that would benefit the customer more than reporting this on paper. That cost benefit trade-off is not an unreasonable approach to thinking about the value of documentation.

Slide 5.28: Optional Exercise 5.5: What “Specs” Can You Use?

Now the notes on page 5-27 point out that, even if you don't have a spec that is a written document titled “Specification signed off boss or signed off customer,” or a document titled “Document Requirements signed by someone of authority,”you still have things that you can use as specifications.

So for example, every time you get a new build you probably get a memo. It might come to you directly or it might be just folded into the source control system -- the list of things that were changed and why they were changed. Those are descriptions of the new behavior of the product and the collection of those specify a great deal about the product. If the programmer who loaded the source back into the system put a comment in and said now I that make this change all mathematical functions should be double precision. And you do some tests and you discover that some things, like division, are still single precision, you have a basis for saying there's an error. And when somebody comes back and says but the original documentation says it is single precision then you say yes, you updated the spec in the source control system. It's not an official spec but it is a description that people expect other people to rely on as to how the program is supposed to work now.

Published style guides: I'm not talking about the style guide that your company writes for itself, that’s a spec. But if you are writing software for the Macintosh, Apple publishes a handbook on what Apple user interface guidelines are. If you publish software for XWindows, there's an XWindows style guide. If you publish software for Microsoft there's a Windows user interface style guide.

And so, if you're in a given platform and you look at the structure of the dialogue and you say this is really confusing, and so you write a report back that says, I don't like the design of this dialogue it's confusing. And someone writes back, I'm so happy you have an opinion, go away. On the other hand, if you write a report that says, I think we have a problem with the design of this dialogue because it will violate customer expectations and customer expectations are described by Microsoft in its book on proper user interface design for the Windows environment and you can see that these are the two standard ways of doing this dialogue, this doesn't match either one.

That's a spec-based report. It happens that this not a spec that you wrote but it is a well-documented approach on what should be (according to Microsoft) normal customer expectations in this market. That's not going to be blown away. People might still say I like what I did better. And you might have the development team say were basing ourselves on a new standard, it doesn't matter.

They're not going to look at you as a fool who has no taste. They will look at you as somebody who has done the research, knows what the relevant spec is, and quoting it for their information. It's much more credit all. There are tons of documents available that were not developed inside your company for the purpose of being called requirements documents that in fact lay out the expectations the users of the software may have that can be cited by you as reasonably authoritative. And that list is in the notes or the list of notes is in the notes.

So you can do spec-driven testing in the absence of specs for many many aspects of the program.

Slide 5.33: Definitions—Risk-Based Testing

We are moving past spec-based testing into risk-based testing. And risk-based testing has at least three common meanings in the test community. I'm going to focus primarily on one of those meanings.

The one I want to focus on is the technical approach. Risk-based testing involves designing a series of tests based on your perception of risk that are optimized to find problems where you think there is a risk of finding a problem. You use risk analysis in the process of designing your test cases.

Another approach that is often called risk-based testing is really risk-based test management. We asked the question what aspects of the program are the highest risks; I will spend my time on those and not spend much time on the areas that might be low risk. There are some techniques that are fairly simplistic for brainstorming, which areas might be high-risk vs. low risk and then allocating our time accordingly.

An example of risk-based testing management is the people to developed this kind of modeling for this program are not competent with visual modeling, we don't trust the code they wrote, so we think we're going to have to spend an enormous amount of time on very complex test looking at all the special cases that they rode into the code incorrectly. That is our prediction, were going to have to spend a lot of time on that. That is risk-based test management.

Another kind of risk-based testing is risk-based project management, which is just project management in general. RUP is the process of a risk-based management system. And a whole lot of your project management's goal is to drive risk out of the project. Your goal to manage the subproject, called the testing project, is essentially the same. Examples of project level risk are your testers are not competent to do the kind of testing that you need to do to test this product, therefore you have risk of not being able to give the information that is needed at the into the project. That is risk-based project management.

OTOH, in risk-based test design, you might begin by saying I don't think I trust this particular part of the program, in fact I think it might fail this way or that way. I better have a test for each of these possible failures. So those are the three levels that we can look and I'm going to spend most of my time on risk-based test design. That's what we mean here by "risk-based testing".

Slids 5.34: Test Techniques—Risk-Based Testing

So, if we think about risk-based test management or risk-based testing the goal stated for both of them is to first go after *the bugs that you are most afraid of*. Find the biggest problems right away.

And so you need to start out thinking about what things are high risk and then figuring out tactics for testing against those risks. When we talked about equivalence classes, we organized things in terms of ways they might fail and then came up with the best representative -- that is the one case that is a little more likely than everybody else to show a problem. And I give you examples like the printer example where I say you know if you're looking for memory related problems test with this printer; if you're looking for paper positioning problems test with that printer.

That's a merging of risk-based test design with boundary instances. It you read papers on equivalence classes, Myers’ book for example, risk doesn’t get described. You have a strategy for breaking things down but you don’t have more than an implicit theory of the error. And in a more modern approach of equivalence classes analysis, the theory of error is made explicit. One is part of the technique is having the theory of error -- how can this go wrong and what is the best test case for going after that.

So equivalence class analysis today is really a merger of the old boundary analysis approach and risk-based testing.

Another way that is management driven, I think I just hit part of your question now, is to use something called failure mode and effects analysis (FMEA). Failure mode and effects analysis is something that is done a lot with hardware testing. Anybody who is doing safety critical software ends up going through some failure mode and effects analysis in the only to be able to check off a list of things that are supposed to do. But were not yet as good at FMEA in software as we are hardware.

Let me describe what happens in the automotive industry for example. Here's a feature for a component of a car its called a steering wheel. Let's do a FMEA on the steering wheel. How did we have a problem with the steering wheel? We need to check for every possible problem we can have with a steering wheel. But our first question is, “How could the steering wheel be bad?”

So we go back to our records because we've made a lot of models of cars and they have a lot of steering wheel and we have a lot of complaints associated with steering wheel. We've also had our own cars that we keep running around the test track, and so we gather a lot of experience with our own fleet and discover over time that certain models have certain weaknesses. For example, the materials in the steering wheel get brittle and at some point in the life of the car you yank the steering wheel and the assembly breaks and now the car cannot be steered. That is a bad problem.

So you write that down on the list, “This part of the assembly breaks /cheap plastic; this part of the assembly breaks/rusted out; this part of the assembly breaks/the rubber got brittle -- didn't handle wetness well.” And so you have this whole list of this with how it failed by the way this was the consequence. “On the track we couldn't steer, we crashed -- good thing it was only going 10 miles per hour. Hate to think what would happen on the street. Lethal risk problem.”

Some others – “the fabric on the steering cracked and had to replace it with a new cover. Cracked after five years -- not even within warranty.” Minor problem but it is cheap materials associated with a steering wheel. You end up listing it and ask questions like how serious is the problem to the customer, how expensive would it be to address this problem in testing, how inexpensive has it been to address this problem in the field, basically -- how much do we care.

Now having developed a list of all the interesting steering wheel problems, we can ask 2 questions”

What would I do to test for this?

Do I have budget to test for this?

Now the "if I wanted to test for this what would I do" question is the test design question and the "do I have budget to test for this" is the test management question.

Notice that we' re not focusing on the general component and stopping there. We're not just saying, “I wonder if I should test steering wheels? If steering wheels were broken would that be a bad thing? How bad would that be? Is it likely for the steering to be broken? Gee, I wonder how to test for that. Maybe I could test for that early.” That level of thinking -- which is often the level people use at the high-level management decisions, should I test component A heavily or not -- doesn't take you down into the failure mode analysis. FMEA is the analysis that enables you to ask how would I design a test for this way for it to be broken vs. that way for it to be broken.

In *Testing Computer Software* we have an appendix that lists 480 (in 1988) common defects in software. We updated a little bit in the 1993 addition, but by today's standards it is horribly out of date. If you want to use that appendix please realize that it is a tiny subset of what you would want to look for in today's products. I have some graduate students in my lab who are working on updates for Web based products. In a year we will have more and we will probably fold what's in the appendix into a different structure that will be very nice.

That structure still won't be the right structure for your company because every company will have different risk and different kinds of information they track about their products historically, different kinds of information available. But if you can get a list of how things fail, go to **bugnet.com**, go to **cnet.com**, go to **winfiles.com**, go find your user groups and their discussion list and start tracking how they complain, go your tech support system and ask to look through all the complaint letters you've received and start getting a list of the many ways your product can fail. And from that list you can say, "How can I test for those kinds of things in this release?"

So build yourself a risk list and test off the list.

Auditing is another use of the risk list. When somebody says, "I can I write a test plan for that," say "Thank you very much." They come up to you with their umpteen hundred thousand pages of test documentation and say, "Here are the tests for this product." If you look through all these pages and try to understand what they have done, you will not understand what they have missed. You just can't keep that many details in your head at one time.

So the strategy that I use to audit test plans that are given to me is that I have a list of common categories of bugs and I sample a few bugs, potential bugs from each category and I say, "Could the program fail in this way?" Well, theoretically it's possible. And then I can go through this big set of test cases and say "Which test case would find this error?" And if there a test case to find this error, good. And if there's not a test case to find this error, I go that's interesting. Go back and find another possible error in this category and ask the question again, what test case would find this error and if there's no test case for that error either, then I just found a category that the test planner didn't think about. And if I'm an outside auditor for a development group, I've probably found a category of error that the development team hadn't thought about which means very high risk.

So you can do risk based auditing as well as risk based test design, but you start from thoughts about how the program could pay fail and then try to figure out ways to find out whether it did.

If we have a great risk list of ways that the product could fail, then we test them in a reasonable order or we look at the nasty stuff before the trivial stuff. How many of you in going from products for example to the help system, go through and look for minor grammatical errors as the first thing you notice? It's very tempting to do. You start marking small spelling mistakes, missed commas, and so forth.

Well you know grammatical errors are important, spelling mistakes are a bad thing --they can be embarrassing -- but if it tells me in the middle of doing something that I should press the nice red button on the side of the computer that is labeled off, that may be a really dumb thing to do. I would rather have that flagged than the fact that they spelled off as "of" instead of "off".

It is more important to ask if there's something here that would lead the customer into doing something that they would really regret. That's a much higher risk related issue. In risk oriented testing you ask the question what would be the most important errors that I can find with this documentation -- let's look for them first. And then we will come back and eventually will get to the commas, but only after we know the rest of the stuff is okay.

Slide 5. 35: Strengths & Weaknesses—Risk-Based Testing

Now there are risks of risk-based testing too. The list of ways the steering wheel could fail might not include the new way that the steering wheel could fail. If you're not thinking about this design on its own terms and are merely going back to history, you might discover that this new steering wheel is made out of a new composite material that doesn't rust, isn’t rubber, doesn't mind water, and isn’t cheap plastic. And you've never had anything that disintegrated after being in the sunlight for three days!

You don't test for what you haven’t seen before. But if you knew anything about materials you would go gee this stuff will really heat up, I wonder if it will last in the sun. You have the problem that if you're doing history-based risk analysis, anything new isn't in your history and so you better check for that separately. You prioritize optimally only if you really have covered all of the risks. If there are any risks missing, you might not even test for them.

We have a coverage problem that’s an interesting piece. If we think of coverage in terms of hitting every line of code, hitting every branch, hitting every basic path, hitting every condition, those are things that we often test and so I achieved 100% line coverage, 100% statement coverage.

In risk based testing you talk about 100% risk coverage. I did all of the risks that I listed as things that I should test for. But if you were tracking yourself that way, what about the 14 you haven’t thought of.

Blind spots happen with risk differently than with code. It's very easy with code to tell when you hit a line are not. You can go to the line, you can go to the test -- did I hit you? The line is there. In the case of risk, if it's not in your consciousness you don't know that it's not there and you don't know to count it as not covered.

So you may be missing risks or blind spots if you focus all of your testing on risk. You may be testing only a quarter of the code. Because you have not thought about the way the rest of the code could fail.

If you're doing risk-based testing alone without asking the big risk question -- *What am I missing?* – you hit the blind spot. Many folks who do risk-based testing don't get to that fundamental question about risk. They address the issues they can think of, in the order they think is important, and then they discover there are big holes after the fact. OK, they won't have those big holes next time, but they'll have others. If you only do risk-based testing then you have this serious problem -- how do make sure that I hit most of the problems and that's not trivial question.

Slide 5.36: Workbook Page—Risks in Qualities of Service

As we think about risk-based testing, we can think of the same quality characteristics that we looked at before. For example, we think to ourselves the risk error is that the program fails to have accessibility. Well what does that mean? Well the program fails to have features that enable it to be read by someone who is colorblind. The program fails to have features that enable it to be interacted with properly by someone who is deaf and so forth.

And so as you look at a quality attribute you can say that I thought a high-level risk came busting out from this attribute, what would that mean, let me break down a bunch of examples, and then test for those.

Take-Home Exercise

An exercise you could do at this point would be to take a product (especially good for in-house teaching take their main product under test) take one of the quality categories and ask the question -- what problems have been found for this iteration of the product, of this kind? How do you plan to test for those things today? If you thought about those things in the category, how many things to you think could come out that I guess you didn’t find last time, how do you make sure those problems won’t come up in today’s version. And if they generate in effect a risk list for one category then maybe this group generates a list for one category, and this group generates a list for another, in this group generates a list for another, you develop the stuff that should fold directly into their test ideas catalog that are useful for their project that there are all in a testing class to figure out how to test.

Slides 5.37-38: Workbook Page—Heuristics to Find Risks

A different approach begins by asking, *Where might I find bugs?* It involves thinking about how the development process might be screwed up.

My favorite example is not on the slide. My favorite example in real life: two programmers in the middle of a messy divorce, writing software that has to interact a lot together – How well tested is that interface? How many bugs do you think they're going to either not found or not fixed? No it's not my bug it's your bug. Don't tell me that. I'm not going to change stop in my code just to deal with your inadequacies.

If you see that dynamic between two people, there's a nest of bugs just waiting to be opened up. It has nothing to do with how the product would normally fail; this is abnormal. But of course it's not abnormal, it happens all the time -- broken office romances happen a lot.

Another common example of a risk is the programmer who has a taste for cocaine. The programmer who has too current a taste for too much alcohol. The programmer whose parents are dying and who is staying up all-night comforting other members of the family. All three of these end up with somebody who's coming to work in a state that's not really fully functional and so that person's work product is probably a risk.

And so now the question is how could this fail? You have to analyze what things they are coding and what stuff that there're interacting with to come up with a list -- anything they do has risk so what possible failures you can find?

The slide and notes list places you might find risk.

- OK it's new technology -- people won't be able to make it work the first time;
- its components you have never touched before and probably don't work right for your stuff;
- it's new programmers that don't know how to code with this language and they probably make the usual mistakes people make who're newcomers to this language,
- and so forth.

All of those have nothing to do with this class of products; they have to do with all of the project management things that go in to make it a product in any class, especially this class. And any of those can get you to say I need to test this work a little more carefully.

Slide 5.39: Workbook Page—Bug Patterns As a Source of Risks

We covered the notion of bug patterns, a list of possible bugs, and I now what to get to the question of assessing project management level risk. How do you decide a testing project is in trouble and how do you manage that?

Slide 5.40: Workbook Page—Risk-Based Test Management

Project management risk is a general question is addressed throughout the project management guidance of RUP. There's also plenty of good stuff at the Software Engineering Institute web site.

You can download course notes and white papers for free that will give you some insights into how some other folks do risk management. Tom DeMarco and Tim Lester write wonderful stuff as well on project level risk management. Brian Lawrence at coyotevalley.com has some other pretty good stuff on controlling projects from a risk case point of view. Stale Amland is mentioned as an example of the best description of the high level risk based test management. His paper is included with the course notes of this material.

So if you're wondering about the time, or lack of time, from managers trying to say I wonder if this area should be tested or not rolling down into how are the ways this could fail. You can see that many of the folks who are operating at the management level never get the detail that allows you to roll down. And yet as Stale Amland very well points out, operating at that level they might still make some fairly good decisions about how to spend their time and money, even though they don't have a list of failure modes.

Slide 5.41: Optional Exercise 5.6: Risk-Based Testing

The notes are targeted toward the public class in this case, where I assume everybody's been shopping at Amazon.com.

If you go to Amazon.com and you might have to have a live connection to the room -- at least one for people to check it out, then you can ask: *What are the functional areas of the Amazon.com web site?*

There are 20 of them --brainstorm them out. As examples we have

- shopping carts,
- credit card handling,
- tracking of customer purchase history (where you come in and it says, "Welcome Joe. Here are recommendations that are based on your previous purchases…" – They have a database and so that feature may be accurate and it might not. You can imagine if they got it wrong you might have a 10-year-old coming in, "Welcome Joe.." – and start selling pornographic magazines – oops right? So there's a certain amount of embarrassment that could come out of a bad parameter.

So anyway you come up with a list of the functional areas and from that you can go one of two ways.

Top down approach

"OK here's a list of 20 possible errors. If you were developing this in your organization, which you think would be the hardest to implement? Which do you think would be the ones most likely to not be done correctly?"

This is Stale Amland's approach. And so we rank, we've actually put them on a scale of 1 to 5, where we say shopping carts never work (I don't know if that's true but let's suppose your group came to that conclusion). Shopping carts never work – 5. And on the other hand, customer history ha, everybody understands how do that database function; it's not a problem it will always be correct that's a 1.

Then you ask the question "OK, suppose it's broken if shopping cart was broken how bad would that be?" They may say it would be average bad, customers with hate us -- this is a 3. Or they might say it would be really, really, really bad -- the attorney general's office and the postal inspectors would come down and arrest us for mail fraud - 5 whatever.

You end up with how likely is it to fail, how serious is the failure that I expect, and in Stale Amland's approach he then takes the two numbers and says if you have a 5/5, test that today. If you have a 1/1, let the customer test it, it's going to work. And when an executive comes in and says, "Can we ship today?" And you say, "Well right now we are in the 3s. Do you want to ship at a level where we won't have tested everything at the 3 level of risk and a 3 level of possible severity?" An executive says, "Yeah let's wait till the 2s." Fine if the exec says that's fine, I'm happy at this point.

You say, "OK I've explained what my classification system means you've made a nice wide-open decision we'll write that on a little note and when it goes out into the field we should expect some probabilities of bugs of a certain kind to show up and if they do don't come back and ask me to explain." In risk based testing in risk based test management, your goal is to make sure that the stuff you tested yesterday was more important than the stuff you will test tomorrow.

And so when the executive in charge of saying we want to ship it today comes by, you know that you have spent your time optimally up to this point and you've justified it.

When you do the brainstorming technique that applies to Stale Amland's method, you might find that people will say, "we don't know enough about the application; we don't know how it will really fail. Basically we don't have the failure bugs under our belt it is too high level and too unrealistic."

One response you can give back is, "Yes, but it is actually used by many companies, good or bad, you should understand the technique because you'll see it. And you should understand that it is not based in failure modes and you should understand what that risk means." You might want to sneak in some testing of low priority areas even though it is not on your list just in case the estimate which wasn't based on analysis of what specific failures there could be wasn't perfect. Sometimes these estimates are guesses that are not well above informed.

Bottom-up approach

The other way you can do the exercise is to say, "Here are features: shopping cart, credit card processing, and so forth. Gee, how could a shopping cart go wrong?"

And some folks will have played with shopping carts and if not, well you have a live connection and you say, "Let's look at the shopping cart - what does it that do? What are some examples of how it could go wrong? What are the functions underneath this broad categorization – let's look at this, let's look at this, let's look at this. This could be broken this way, this could be broken this way, this could be broken this way."

You brainstorm up a list and you come up with a pretty big list pretty quickly and you say, "OK now we can go after each of these possible errors." And the neat thing about this is that you can take someone who is a reasonably good tester, who has never done web-based testing, and have them finding bugs today. Take what you know about how things can fail, get some information about how Web things can fail, and merge them with how could this one fail and go check. And as you discover the order didn't fail in this way, you discover more about the product and you come up with different theories of other ways it could fail and could run off of those.

This is specifically *exploratory risk based testing.* The first wave of work might familiarize you enough with the kind of application you're testing, so that you could go through and do a very systematic set of test documentation after that to guide the rest of your testing. But coming up with the sense of how risky the product is might be exploratory even in an organization that intends primarily to be formal in its approach. To the extent that you don't understand what you're testing, you will be exploring for awhile either intentionally or in spite of your process.

Slide 5.43: Test Techniques—Stress Testing

Stress testing. (The word stress testing has too many different meanings. Like so many other words what is a test case, what is a test plan, everybody has a different definition of what is a test.)

Here we use "stress testing" to mean a determined attack on the product with the expectation that you will drive the product to its knees, in order to find the first place of vulnerability in the product so that we can fix that.

The first place I saw this technique was a company, where several people had congregated after leaving the company called Tandem. Tandem in those days specialized in doing fault tolerant computing. They expected that really bad things would happen to your computer and your records would stay fine. So banks would buy Tandem computers, because they were confident that their bank data would always be safe. And you would like your bank data to always be safe and current.

So here I am hanging out with some people who had been at Tandem and they talked fairly reverently about one other person from Tandem. This guy talked about how he tested the fault tolerance of storage system. He describes a test where he'd be writing data to the database and he'd walk up to the disk pack -- disk packs in these days were big machines -- and he would open the cover and rip the disk pack out while it was being written to. This might destroy the disk pack. The interesting question is: *Has the data been successfully updated somewhere else? Would the end-user even realize except from an error log that a problem had happened and was addressed, or would it just go smoothly?*

So he would do that and make sure it would go smoothly. And he described a few of these tests and then he described one of his favorite ones. One day, he went out to the rack a got a really complex bunch of stuff happening and he grabbed the rack and did this {rocked it} and then did rack-foo…BANG. Needless to say not everything survived. Amazingly his job survived. People talk about this like, "One day he did this, wow, tell him about that one."

Why would experienced people that I respect have respected this guy for just being a barbarian? Well, what he was doing was subjecting in his system to greater and greater stress until eventually it did fail. But when he was also tracking was every process that was involved, so that when it went down, in this case literally, he could see messages as to what failed first and given that that failed, what failed second. And then he could go back and he could ask the question,

Is it inevitable that this will fail given this hardware event or is this just a weakness in our imagination of what could go wrong? Should *this have failed or shouldn't we be tolerant about this kind of fault too? And if the first one failed, and the second one now fails is it inevitable that the second one failed or is that just because the programmer involved in the second function made the critical assumption well, the first function would always work and didn't protect themselves against a predictable risk in the system. Do we need to harden the code?*

And so we put the code under tremendous stress and discovered which pieces needed to be hardened.

If you had asked this person why didn't he do normal testing like looking for boundary cases? Why don't they do regular testing? Why are they going out looking for problems that would almost always be deferred? Imagine a bug report, "If I took an axe to the computer, destroyed everything, and the program didn't work…" Can you spell P4 -- not a bug go away? Work for our competitor please, we'll arrange it.

So you can ask people who are working on this kind of testing, why do you do this approach, why don't you do normal testing? And the answer from some of the most creative of them seems to be,

That should have all been taken care of by the programmers. Why should we sweep up after programmers who, if they were competent, would have taking care of things like equivalence class analysis testing anyway? That's not where a skilled tester should spend their time. This is interesting, this is stuff you can't see inside of the code -- you have to look at the whole system. It takes an expert.

These days we have whole groups that are doing testing like this. Load test tools have just become over the last few years discovered as reliability test tools. You drive the system through way too much traffic and you discover that there are parts of the system that shouldn't be failing but are. And it is better to notice that during test or during early deployment than later.

So more and more, this approach to testing is becoming in some companies a dominant approach either of the company or some subgroups of the company that become experts. Now in terms of the things you're blind to, and anything you cannot expose with stress testing, you're not going to see this way. But this will expose problems that the other kinds of analysis we looked at wouldn't touch.

Slide 5.46: Test Techniques—Regression Testing

Regression testing is the repetition of tests, often automated, after changes. It's important to understand what variation is built into these tests.

Invariant regression testing (same sequences, same data, same options, same configuration) is useful in a very narrow set of circumstances. (Some organizations found this very valuable for Y2K testing, for example.) The problem is has been called the pesticide paradox (Beizer), the minefield paradox (Bach) or the immunization curve (Kaner).

In short, you immunize the program against a set of predictable tests. To the extent that the program has passed exactly this test 20 times already, the degree to which you might expect to get information from the 21st run is not necessarily very high.

On the other hand, you can use a set of similar tests with variable sequences, variable data, variable options, or variable configurations to look for similar kinds of risks. This can be a very powerful use of test automation.

In either case, maintenance of the tests is a vital consideration. It is important to consider the test automation in conjunction with the testability of the software under test and the capabilities of the automation tool.

Slide 5.49: Test Techniques—Exploratory Testing

Now we have exploration. I've defined exploratory testing a few times. Basically you're in the mode of learning, testing, and interpreting all at the same time. There is a document the got put together called, The Software Engineering Body of Knowledge, some people primarily in the Institute for Electrical and Electronics Engineering, some people think it is a collection of best practices for software engineering and you drive your development of software engineering university curriculum and would probably be the basis for licensing software engineers. Amazingly they talk about exploratory testing and say that exploratory testing is the most common form of testing used today and then they say that it should only be done by experts.

Well, I know that we think we're doing non-exploratory testing many times. You analyze a spec very carefully, you predict a set of results, you derive specific details of the test cases from the spec, you run the test cases and see a failure, you write up the failure. It doesn't seem like very much of the exploratory there and it isn't until you try to troubleshoot the problem to see if you're looking at the tip of the iceberg or if there's something worse. And a funny thing about the spec, the spec tells what the program is supposed to do if it works. It is not much in most specs that say it's supposed to add two numbers but by the way if there is a bug here then it will fail in this way.

And so what you find out it doesn't add two numbers, there is no spec to go to that tells you well it's error is really constrained to this circumstance you don't have to worry about it except under here or here. It's not in the spec. Instead you have to troubleshoot the bug and say how big is this problem?

Welcome to exploratory testing. Every time that you competently research a bug, you're doing exploratory testing. You hand in a bug. It gets fixed. And you take that report and you say I wonder if it's really fixed? You run exactly the test case you ran the first time, most testers don't stop there. Gee, I wonder if a variation of this test would get around their fix. You write up a report that says you're adding two numbers and I tried that and I tried 2+3 and it gave me 6. Well, there's a useful way to fix that and some programmers actually have done this, they write a routine that says if the first number is 2 and the second number is 3 write 5. That's the most extreme.

They write a special case that gets around your bug report. OK, all you have to do to find out if they did that is to write some other pair. Stunning! There have been some cases in DOD software were the contractor gets charged eventually for in effect fraudulent fixes where the product went through a whole lot of iterations where nobody ever realized that these Mickey Mouse fixes - if the first number is 2 and the second number is 3 print 5 - were just that because they only tested the specific case that found the problem and that they reported. They didn't do any variation testing.

You're not going to make the mistake. You're going to see if there are side effects from this fix. Every fix carries two risks - it didn't fully fix the problem and it broke something else. Of course you're going to ask the question, what else could have broken? And at the only fixed part of it, what part might have been fixed and what part might still be exposed? Welcome to exploratory testing. You don't want to do that, maybe you don't belong in a testing group. You're hearing some emotion because I get stunned when I see people who are talking about testing say only really skilled people do exploratory testing. Only people I wouldn't hire in a test group don't do exploratory testing because any aspect of analyzing a bug is going to involve some exploration. Exploratory testing is testing with your brain fully engaged and I like that in my staff.

Slide 5.50: Strengths & Weaknesses: Exploratory Testing

Exploratory testing has several strengths. Apart from the fact that it is the dominant approach used by testers, the other advantages are that it recognizes a reality -- you know less about the program when you start testing than you will know as you progress through. Imagine the risk associated with writing all of your test descriptions at the start of a project when you are at your greatest ignorance about what that product is and how it could fail. Even though you might get, in the best of all possible worlds, a complete specification that is accurate. That specification still doesn't tell you how the program will fail. It describes how the program should work. But what you will find is it does fail. And what you will develop will be tests that exploit the weaknesses in the program as coded not in the fantasy of the specifier's head.

You aren't trying to prove that the program works according to the spec because it doesn't work according to the spec. If you think something is wrong with you and you go to the doctor and the doctor says, "Everything's OK." Your first reaction is probably not "Oh, I'm so relieved!" It is probably, "Why did I sign up for an HMO?!" We are not trying to be the HMO of software development.

To be good diagnosticians means we have to understand what failures look like. And the failure pattern associated with a program reflects individual characteristics of the individual people who make mistakes in that program. And the individual trade-offs of what they demand of the stakeholders who now face limitations they didn't expect and who now say this is what I want if I can't have what I was suppose to get.

That evolution goes on throughout the whole project. That's why you have an iterative lifecycle in RUP. The idea is that you would go into testing and create something like a waterfall. Let's plan everything in advance and then test. It's just as insane as the idea of trying to beat up for the code. It will not work. We've seen it fail many, many times, and the sweeping under the carpet the fact that we have a highly creative activity is not a wise thing. Now there's a kind of testing that some people call exploratory testing which I call testing by idiots. Testing by idiots works like this: I don't know what this program is supposed to do and I don't care; I think I'll just sit down and pound on the keyboard. And

So when you ask somebody what they mean by exploratory testing, they say oh yeah I give it the shoe test. The shoe test is where you take your shoe and put it on the keyboard and it sits there and just pounds a bunch of keys. And they say that's the method. It's really good, it's generic, not every program can handle the shoe test. Well of course, input overflow test are just one tiny mean being that you might do. But if somebody brags that this is what they know to do to test the program – well, my daughter did that when she was testing code for me when she was 6. She used to come out and visit. I paid her two bucks a bug. Some people have allowances. I gave her little testing paths. She found bugs and then went off with her crispy little bills and earned what she then bought for herself. It was a nice kind of thing. She can do shoe tests today. We can do shoe test too. But when somebody tells you that this is their notion of exploratory testing, they're talking about a 6-year-old's version of testing.

Exploratory testing involves gaining a deep understanding of the program and its risk overtime and tailoring your tests to the situations that you face as you face it. That is the essence of exploratory testing. It's not easy, but it is what every tester is trying to do.

Who should do exploratory testing?

If you're going to do that I'm just making a note here. The full point that I said is

Not everyone is a good explorer and what you should avoid doing is forcing someone beyond their confidence, their comfort zone. People who are very good at following a systematic plan as intended, and very good observers that need to have things either planned out by themselves or planned for them, and then execute that plan are not necessarily happy or good as explorers. Similarly some people who are great explorers feel very constrained with excessive written documentation. In a diverse group you want both populations and they will help you do different things.

Slide 5.52: Test Techniques—User Testing (1/2)

I think we basically covered user testing.

The notes mention that there are several different examples of beta testing. Beta testing is one of the most common forms of these tests. There is a particular kind of propaganda that the executives in some companies have heard about beta testing. And that is that beta testing is such an efficient way to find bugs that you can do most of your testing by foisting a defective product on your customers pre-release and getting the reports back as to what went wrong.

It’s not all it's cracked up to be. There are certainly folks who wander through the field as consultants who claim to have been involved in some very famous releases, which were allegedly heavily tested through beta testing and not much testing beyond that. I have heard consultants who claimed to be educated testing consultants who are primarily project management consultants that say an extensive beta testing program allows you to cut your testing staff in half. This is a wishful thinking.

It never happened that way. Some of the classic projects that they talked about as having had their risks managed by beta testing are projects that I actually had involvement in and from the inside that's not how it looks.

I have a nondisclosure problem but let me just describe one project in an anonymous way as the project that distributed to many, many users and vendors pre-release... The manufacturer of that software had great confidence that their software was compatible with a wide range of products based on an absence of beta criticism including calling many people up and saying, how’d it work? And hearing people say, oh it worked just fine. Now what many of those folks actually meant was I have yet to open up in the shrink-wrapped package that you sent me but I think I'm not going to admit that over the telephone.

The manufacturer of that product found out about a week before the scheduled ship date that one of the applications they were certain was fully compatible wasn’t. It happened that the lead tester for that part of the product had a personal copy of that and discovered to her shock that the thing crashed under the fairly obvious circumstances.

That company not only slipped its ship date but ended up going to a software store and buying over 1100 different products and testing the thing in-house. The slip of this product was attributed to a whole lot of the reason but one of the two main factors behind the slip was that this beta program like most beta programs was not well designed to find defects. It was very well designed to find out whether the product was acceptable to the market. If you have something that has to go through a lot of networks for example you want to have deployed an early version to gain some confidence that the thing will be well-behaved when they finally buy the final copy and put it on. But if it's not well-behaved you can’t count on those people to explain to you in a competent way a) that there's misbehavior and b) what that misbehavior is.

Slide 5.53: Test Techniques—User Testing (2/2)

User testing unsupervised is an extremely hazardous thing to rely on for liability information. If it is specific information that you need to get from your customer sites because for example you can afford to replicate all their configurations, create test materials that will allow you to remotely tell whether that program is passing or fails.

There is a very simple example of something that we did at Electronic Arts. We made many programs that printed in very fancy ways on color printers. We gave you the files to print as part of the beta, you made print outs and wrote on the back of the page what your printer was and what your name was. If you were confused about the settings, when we got your page back, we called you up. We had a large population of people with a large population of strange and expensive printers that we couldn't possibly afford to bring in-house. But we knew what the output was supposed to look like and

So we could tell whether it passed or failed. We also did things like sending people parts of the product and a script to walk through and we would be on the phone with them and say what do you see on the screen? We wanted to do video compatibility where they’re across the continent

So you are relying on their eyes to be your eyes. But you’re on the phone, you don't ask them if it looks okay, you ask them what is in this corner? And you structure what you're going to look at

So that he can be informative about the kind of bugs that you think you might find with video incompatibility. If you think you are at risk on configuration you should have some sense of how configurations will show up the configuration failures. Write tests to expose those, get them to your customers, and then find out whether those test passed or failed by checking directly with that specific test. That will budget's situation where you can have some confidence. But unsupervised beta testing while it is very good, a mass unsupervised beta test is very good for marketing purposes. For early deployment purposes it is not a reliable method for assessing reliability find bugs.

Slide 5.55: Test Techniques—Scenario Testing (1/5)

Now we hit scenario testing. Scenario testing within the Rational Unified Process has a very specific meaning and I will hit that. I'm also going to hit a different very specific meaning that comes out of user based task analyses that are not used in RUP.

Now, common to both of them is the notion that we are dealing with things that are expected to be realistic. Common to many of the scenarios as in the Rational world and all of the scenarios in this other world is often the notion of complexity. It's not just a simple one-flow thing. It is an end-to-end task that gets done that involves probably many use cases. And we'll see that.

One variant of the scenario testing world is called soap operas. And we'll take a look at Hans Buwalda's description of scenarios as soap operas. It's interesting to see the kind customer story that Buwalda can come up with and how much his stories create very effective ways of getting folks to understand the realism of the test cases he's working on.

Slide 5.56: Test Techniques—Scenario Testing (2/5)

Let's start out with use case based scenarios. In a use case you have several flows through use case you have the normal path through - what the person is trying to do - and you take that path, it's the happy path. You want to make a phone call you start, you dial, it rings, you say hello, you talk as long as you want, you hang up. Everything connected. Everything worked. You smile and say, oh good that's what I want - I'm happy. Now we say OK in this start-to-end telephone case what could've gone wrong? Well there is the normal case "gone wrongs" like nobody was on the other end, you called and nobody answered; you started dialing but forgot the phone number and hung up.

So there are paths that are driven by what else the user can do in that situation and there are some other cases where we say, well what if the system had a problem? And you called but the other phone was having some software problems or while you were calling someone came in and tried to make a call to you. All of those end up being basically complex more or less variations on the happy path.

If we apply values, I don't mean values like good and bad, I mean the values like 555-1212 - that's the phone number were going to dial. I'm going to call at this time. I'm going to wait for this long between digits. I'm going to have this phone generate an interrupt to my phone when I try call waiting. If I apply values to make it something that actually gets done I've gone from a use case to scenario. I've got substantiation of the use case.

Now a scenario can be complex or can be simple. If I go through a simple phone call and that's all I do, that's one thing. If I go through the system and I say I wonder what's going to have been if I sit a busy stockbroker at this phone and tried track half a day worth of calls through the system. We are putting the system in effect under telephone load - everybody's trying to call this person this person is trying to call out to make lots of deals. He's got two phones, he's juggling people back and forth, back and forth, hang up and dial again. Constant odd things, we'll create conference calls, we'll have long holds, we'll transfer things for one person to another, we'll end up with a day's use.

That might still be just one scenario. But way more complex than one use case. We are asking in this case, how does the system look in certain circumstances over time? That's a valid question to ask. We can construct that by saying that would be a concatenation of many use cases that come into play. What is the use case for transferring calls, what is the use case for call waiting, what is the use case for putting people on hold, what's the use case for trying to carry six different calls on hold if you have that kind of the telephone at the same time bounce back and forth between them, you can have every single one of those things modeled as a separate feature, a separate transaction but over the day we have as a random ordering of many of them some imperative. That can still be one super use case and it certainly is one kind of scenario.

Is there a difference between a good use case from the standpoint of requirements solicitation and a good test scenario? There is a difference. If you don't think about that difference well and you say, "Oh I'm testing all the use cases," you end up with much less test coverage than if you think about that difference thoroughly. "these are the scenarios I'm going to test because these are the ones that actually represent a day in the life of a stockbroker." OTOH, if you wrote your requirements according to a day the life of stockbroker that way, you would end up with things stated much more complexly than they need to be. You might tease out of those atomic use cases defined as these sets of flows that provide some value to somebody, but you probably wouldn't. And the point of this material is to get exactly for Rational customers to the heart of what's the difference.

Slide 5.57: Test Techniques—Scenario Testing (3/5)

Whether or not in a given company there is an agreement within the company called the Rational Unified Process it's still the case that use cases may our may not be well done. They may or may not be done all. And the code will come to you in the state that it comes to you. The process will be what the process will be for listing the requirements and developing the code. And you will be expected to help the development organization make the product better. Whether you have useful use cases or not.

To this point we have focused more on the program and the components of the program and the things that the program interacts with and less on the way that people use the program. If the requirements analysts came up with great insights about how people use the programs and documented those in a way that was useful to the testers, that's wonderful. And if they didn't you still have to come up with the set of test cases that ask the question, "Is this program worth using?"

It's not good enough to have a program that doesn't accept bad data. "Here, I'd like to pay $1 million for a program that doesn't accept bad data." "Oh good what does it do?" "It doesn't accept bad data." Would you pay $1 million for that, probably not. You want to pay for a program that gives you a benefit and it gives you benefit under the real-life circumstances under which you're going to have to use it. And those real-life circumstances are not one feature at a time. And they aren't one little stratified sample at a time. They are under very multidimensional situations, we want to get something done in the world and we have to accomplish it. Scenario testing is focused on that.

Many of the folks that talk about scenario testing say, *Look, I'm not interested in the customer's first-day of experience with the product. I could come up with simple scenarios for that. But we probably hit those with the very basic features test anyway. I'd like to understand how the customer's going to feel about this product six months after they start using it when they're good at it. Can they make it do what they bought it to do?*

And to test for that, we end up running scenarios. These are complex tests. The four characteristics that make good scenarios are:

1) The first is realism. Realistic means that you can credibly say that real humans who are within your target market would do this. And I will relax the target market for a second because we can have security scenarios where we can say real humans who could get access to your system would do this and probably will try. Is this a credible case if you present it to the development group or marketing group they can say yup I believe this case.

2) The second piece is ease of telling whether it passed or failed. This is one of the huge failings of complex tests cases as they've been done for many years, especially complex automated test cases. You end up running a bunch of tests so that it is so hard to tell if the program passed or failed that you end up saying if the program ran it must be okay. And there have been several disasters of the form: "We ran the test. We thought everything was fine. We didn't realize it was corrupting its input data. We didn't realize it was calculating the wrong estimate for a construction project and somebody is now going to build the building and underbid by 1 1/2-million dollars because everything was formatted well, everything was plausible, but it subtracted some numbers instead of adding them - oops." If you can't tell quickly what the right answer should be you're not going to do a thorough check. And in that case you get a feel good from writing the test which is more dangerous than not running the test at all. It has to be easy in complex testing to check for results.

3) The third thing is complexity. We're not asking how simple things work. We're asking how this thing works when you really try to get the program to show off what it can do.

4) And the fourth thing is the stakeholder. There has to be somebody in the company you can appeal to if you took this failure and walked up to the lead programmer and said, look at this. And the lead programmer said, "Yeah, so?" There has to be somebody in the company that you can walk to end say, "Do you know it does *this*?" And that person would go, "Oh no, that's an important problem!"

   The reason that the fourth criterion is there is that a good scenario test might take you ten design days just to put together to configure itself and run. It might take you a huge amount time. Imagine a tester running in after working on something for ten days and getting the system to clearly fail and the entire development team looking at it saying, "We don't care." Worse, that's a huge waste of company resources on something that turned out to be irrelevant. How demoralized do you think that person is going to be? It is better to ask yourself while you're designing the test, “Who will advocate for this bug if I find it?,” than to ask yourself after the fact, “Why won't anybody advocate for this bug?” If you ask yourself before the fact, you might design your test just a little bit differently to tweak the imagination of the specific person that you have in mind. Maybe they’ll let you use some of their data or run it through their department’s computer and watch it crash. Whatever, you have to find somebody who will advocate for the bug.

So those are the four characteristics. You might derive this through use cases but you might not. But for the kind of advanced scenarios I'm talking about these are the common attributes that up saying the company after company that base themselves on the sky testing.

Slide 5.58: Test Techniques—Scenario Testing (4/5)

Now the core risk associated with this style of testing as the dominant style is late discovery of defects. By the way, let's think in terms of the sequence of events that a healthy test group uses when they do this. New code comes in -- they start with function testing. Why do they start with function testing? Because you wouldn't want to do something like scenario testing to start.

Let's imagine a what scenario testing would look like at the beginning of testing. The program has 100 functions, 30 of which work. You start your scenario test with this combination of 70 functions. It fails on the first-line, you write about it. You go to your next scenario test. It fails on the second line, you write about it. And then you wait while the programmers fix their code. Because their first bug is a blocking bug it's going to stop you from testing. Why is it going to stop you from testing? Because all of your tests assume these particular features will work and they don't.

Great, they fix that bug, nice programmer. Now you rerun your scenario tests. They passed line 1. They passed line 2. And now they crash on line 3, when they hit the next feature. Repeat for six weeks. At the end of six weeks, you finally get to the output features. You've been on input, basic calculations and they all fail, and you're finally getting to the output features and they don't work either. But you're beginning to get to the end of the time allocated for testing and you're finding serious output bugs way late in the schedule. Why are you finding them then? Well we couldn't get to the features until we got past those other bugs. Why didn't you get to these features? Because you structured your tesst so that you couldn't get to output until the input worked.

Scenario testing sets you up for that kind of a problem. Function testing isn't. Function testing is designed so that you can get anywhere, not with complex tests, but you can get anywhere. At the point that anywhere you can get is at least superficially okay, *then* you can feed it more complex tests. It must have all of the features working for you to see the more interesting combination of problems. The risk of scenario testing is that you are delaying the ability to even look at many other critical aspects of the program until what may be the last few days of testing.

Order of applying test techniques

So normally people will start with function tests and they'll extend their function tests into things like equivalence-class tests and/or specification-based tests, and/or risk-based tests depending on the kind information they have and what the intellectual orientation of the people in the room is. And they're now looking at slightly more complex things, but still units of some sort, that are fairly well specified, even though there are many features that may be interacting together. At some point they'll say, "We are up two or three levels of complexity past function testing and the program is still looking stable, now let's make it dance."

That's scenario testing – "Let's make it dance." Scenario testing often comes fairly late in testing. The problems that show up are very often fairly complex to solve and possibly very dangerous, but they are problems which could not reasonably have been exposed before. One of the huge complaints about scenario testers face when they finally show that under certain circumstances you can corrupt the company database and so forth is, "Why couldn't you have found that sooner?" The answer is that the code was never at the point where it was stable enough for us to get to something this hard to find, or this complex. Even if you could have found that specific bug if you knew exactly the path to get there, the code wasn't complex enough for you to reasonably follow this kind of strategy until a certain point.

Now another issue with scenario testing is the coverage problem. In organizations that rely on transaction flow testing which I'm going to loosely equate to scenario testing, basically on business case testing, it is common to see reports that when you hook up the coverage monitor to ask what lines of the code have been executed, it's common to see reports that these tests groups have hit 35% of the code and missed 65%. How can this be? You do this wonderful business analysis. Well part of problem is there is lots of error handling that is not necessarily covered.

If you're thinking about what customers would do when they're trying to do something reasonable that you want them to do, you probably are not looking at what happens when the janitor knocks the Ethernet cable out in the middle of an attempted data transfer. That is not a typical scenario test. It's back in risk-based testing with simple risks. But even if you say what are all the interesting things that we would like to test, for example, what are the interesting use cases? If you develop a bunch of scenarios and then gave yourself a traceability matrix and said here's use case 1, use case 2, use case 3, or feature 1, feature 2, feature 3, depending upon what you are starting with a work customer action 1, customer action 2, or requirement 1, requirement 2, etc. and you hit your test cases, here is your scenario 1, scenario 2, and so forth.

If you develop a great set of scenarios without being very carefully focused on coverage, you are still likely to miss many of the things that you think you should hit. So, for example, I got to interview Hans Buwalda, one of the masters of this kind of scenario testing, and ask him, “What about the coverage problem?”

He drew a chart like this. He said, “When we get all of the soap operas together (soap operas are a specific kind of scenario), we map this way we find that we hit about 65% of the things that we really wanted to test.” And I said, “That's a lot better than 35% - how you get to 100%?” He said, “Now that we see what we have and what we don’t, we have to make artificial cases to cover the rest because we just haven't thought of great scenarios to cover them.” If you're not conscious of that risk and you're relying primarily on scenario testing, then you are going to miss important tasks that people do that you need to test. So coverage is the other core risk with scenario testing.

Slide 5.59: Test Techniques—Scenario Testing (5/5)

Let me tell you a little bit about soap operas. I corresponded with Hans, who is pretty well known for data driven testing, and some of you taking this class might have read his stuff on guidance for well thought out automated test parameters. I started corresponded with Hans in the mid 1990s about that and I finally met him at a STAR conference and after we chatted a little bit about automation he looked at me and said, "So do you ever watch soap operas?" And I looked at him -- I didn't know him very well at this point -- and I said, "Hmmmm, not very often, why, do *you*?" And he said, "Oh yes I love them." When you're meeting a new person at the start they have a little mirror, a virtual mirror, on top of their head that is the strange mirror. And the strange mirror was going "Ah, strange, yes."

So I looked at him and said, "Tell me about this, Hans, why do you love soap operas?" And he said, "They are such wonderful slices of life." And I'm wondering if he's been out of the hospital for long -- what happened that got him to think like this? He said,

> *No, you might not understand. A soap opera takes something that might actually happen and then puts it into wild combinations. It's just like testing, they are test cases of life. Look at the strange things that happen. Somebody gets married to somebody else and it turns out she's his sister. He goes off and commits suicide, at least everybody thinks he did, but two episodes later he's back. No it's his twin brother, his looked like to twin. They said they were not genetically cloned they can be married. And on we go through this ridiculous thing and somehow people will watch this thing and think it is believable because the individual pieces are so close to things someone has seen that you can suspend disbelief about the rest.*

And I said, "Well, that is kind of a fascinating perspective. But what does this have to do with testing?" "Oh," he said, "I test programs this way. I write soap operas." And I said, "Tell me about it." One of the several examples that he likes to use involves a pension system.

So Joseph gets married. He's with a company who does business all throughout Europe and they offer a pension plan and so his wife is part of the pension plan and they have a child and then they get divorced and then he adopts a child and then he gets married to someone else and they have more children and then she dies and then his first child dies and then another one of his children gets disabled and then he gets married again but by this time he's moved to another country -- same company but another country -- and gets married again but shortly after that marriage this company is taken over by another company whose head office is in a different European country... This goes on for a few more years and he gives the details of the years until he says, *OK, what share of the pension plan is the second son of Joseph's third of wife entitled to?*

Welcome to a soap opera. You see this complex series of life events and you say every one of these things could happen. This is a realistic pension situation. This kind of complexity goes on, people move around inside the European Community, companies get bought, companies get sold, pension plans change, insurance companies come and go, children come and go -- we won't even ask about the one he disowned -- but that kind of stuff happens too. And then you have a system that has to calculate who gets the benefits. This is a really hard test. But if you walk up to an executive in the pension fund and ask could this happen and? They go, "Oh yes." "Should the system cover something like this?" "Oh yes." And then you walk to the programmers and say, "Do you code for this?," and they go, "Uhhhhh, sure..."

Now we check. It's complex. It's realistic. If you work with the right people managing the fund, the business manager, you can figure out what that person's share should actually be, it's predictable. And somebody will surely scream if the system can't do that calculation. But it's not at all obvious walking in whether this system will be able to handle that task or not. That is an example of a soap opera.

And if you had covered with artificial data-- let's imagine somebody who had three wives and 17 children, disowned two, 6 died at these stages, moved countries three times, there was a buyout that changed ownership -- let's run the test. People would probably say, "This is a corner case. This is a strange artificial set of data -- what did you do a random number generator? -- we don't have to look at this." But if this is couched in this person who moves the around and has these life experiences, people go, "Yes, of course, that could happen." That's the soap opera. When somebody sits back and says, "I can recognize myself and my friends in this, it's a little exaggerated, but I can recognize who we are. I can see this." You get the soap opera notion. It tells a story every time. And it's a story that is believable or almost believable. Believable enough, that someone could say if this didn’t happened exactly like this, something close to it could, so we better be able to handle this one too.

Whatever your underlying model is that will help you understand the components of the soap opera and help you predict the output of the tests, the goal of coming up with things this complex and checking the system against them is constant. How you derive the details of the test, depends upon what kind of information the programming team can give you, what kind of relationship you have with the SME, what kind of domain expertise you have in your testing group. But the notion of that class of test (and you certainly want to test the system against that class of test cases), it takes a huge amount of creativity and time to come up with winner cases like that and that’s the essence of scenario testing.

You end up with a cost/benefit trade-off where you ask, “What would the cost be to generate a new scenario and the information value of the new scenario compared to the maintenance cost and information value of the old scenario?” Really the value of a test case lies in the amount of information you get from it. If you run the test case and it passes, that is still informative, if you didn't know whether the program would pass or fail it when you run it. But if you run the test case and you say I don't even have to run this I know it's going to pass, you just wasted your time. To the extent that a test case has the potential to educate, it's a powerful test case -- more potential = more power. If you believe that from version to version, or year to year, the program has had significant risk of not passing old scenarios, then the old scenarios need to be kept around. OTOH, if you believe, that once the program has been tested so that it passes this one and it is likely to handle that case from year to year, then that case is less interesting and it's not clear how much work is justified to keep it going. And I can't tell you what it means because the system is very tremendous in terms of long-term information potential for identical cases. But that analysis has got to be done. What's the price of the information you're going to get?

Slide 5.62: Test Techniques—Stochastic or Random Testing

The last one we are going to look at it is random testing. We are not going to look very hard, because this is just too complex to teach in this level of course in this time. We do have a paper in the notes called *Architectures of Test Automation*.

Let me explain the notion of the stochastic testing. Up to now we have looked at lovingly crafted, individual tests. The most crafted of them were the scenario tests, but even the function tests -- we turn our brain on and we say let's look at this little thing, one thing at a time. Let's look at the strength of each of these approaches.

Function testing - the power of function testing is every single thing you do hammers at an individual function. A great function test is one that really attacks an individual function and gives you insight into whether that function is working or not.

A great equivalence class test is one that represents a large population of test that you would otherwise have had to run but you have a perfect representative that can assure you that this class of error it that there will be shown.

A great stress test is one that has the power to show you what things might need to be hardened.

A great exploratory test is one that takes an increasingly deep understanding of the system and translates it right back into attacks of some deep part of the system where you don't yet know what is going to happen.

A great scenario test is one that is absolutely believable but it might be traumatic for the program once you try it out.

All of these assume some pretty careful thinking. That's great.

But the main testing problem that we started with ist that there are a gazillion tests. We have to pick the 5 or the 20 that will represent that population. We talked about automated test series of 10,000 test cases a lot of tests maybe 100,000 tests, it's a lot of tests, an entire regression suite. *Which*, compared to trillions times trillions times trillions of possible cases, *is trivial*. I see the analyses of some moderately complex programs that are into computing the number of test cases that could be run - greater than 2^{90th}. This is a larger number than the current estimate of the number of molecules in the universe. This is all lot of tests that you can run. 100,000 is not even in proportion to a grain of sand compared to the MassPar computer.

So instead what the random tester says is, *We have to speed up all aspects of testing to increase the number of test we can run by many, many times -- let's not think of thousands of tests let's think of billions. To do that of you have to take the human out of the design, out of the code, out of the explanation, out of the evaluation, because everything a human touches is slow.*

So what the human does is scratch and then the computer executes on the scratch. Now there's a problem with machines and that is that they're pretty stupid. You're not going to see machines generating believable scenario tests. It's not what they're going to do. But for instance, pretend that you are testing Excel 2004 against Excel 2002 to ask the question, are the trigonometric function in 2004 still basically the same as the trigonometric function in 2002; have we introduced any new miscalculations? You could do that with a few hundred tests of algorithm, sine, cosine, and so forth. For you could use 2002 as an oracle and just have an execution program, robot, generate a random number take sine, generate a random number take sine.

How many successful comparisons with it take for you to say I think these two functions are the same? There's got to be a number and its got to be finite and at a certain point you're going to say this machine ran for two weeks comparing all the trigonometric functions didn't find any different, printed out every hundred thousand tests from 2004 equaled 2002. You're going to say, "Yeah that's what it's supposed to be." You're going to look for that just in case there is corruption in the system at some point over time - you're going to have weird numbers. But you plan out every hundred thousand tests and what they got an after a few billion test you might say find I trust them.

The human time in this would be the time it took to set the system up the arrest was automatic. But if you find a bug there's more human time but to reserving the person for investigating the clear think that indicate something is broken and having everything else done by machine that enabled a lot higher grade of testing. The only problem is there's only a narrow range of circumstances where you could pull it off. Having an oracle you can trust is one. In general having high-volume automation strategies ask what can we know about the program and what almost fully automotive strategy could we use to exploit this knowledge.

I promised we'd talk about is monkeys. The notion of monkey testing comes from the statement that if you had one million monkeys with one million typewriters and they typed for one million years one of them will type up the writings of Shakespeare. And so if you can throw one million testers at a program and have them test, eventually they would find all the bugs.

There is the dumb monkey high-volume test generator. When Noel Nyman started working with a certain test generator he called it a dumb monkey, in honor of the claim made by certain testing theorists that dumb monkey testing would never show anything of value. The notion of the dumb monkey is you get to a state, you can apply your random number generator, and you can't say what is next. And that might be a valid next and it might be an invalid next it's whatever it is on the map. But you generate basically are random sequences and the program does whatever is added to and you generate your next random input, and your next random input and at some point the system crashes.

Now in the very simplest form, you have what Noel calls an executive monkey - one that is content to randomly pushes buttons until something breaks. A dumb monkey is generally a little better behaved. You can constrain the monkey in a little better way -- for example, you would never allow access to the mail system on your computer. One of the things that we can guarantee in probability theory is, that if there is any statement system you can get to, a random process executed long enough will get there. If you don't want it to format your hard disk, don't let it get here. If you decide "Oh, that would never happen." then it will.

So we form constraints around the things to safeguard your system and your reputation and execute until it fails. And then we use whatever tracing capabilities you have to try to figure out what was wrong at the point that it fails. That is your dumb monkey form of automated testing. T

he variations of that people try on the one hand people put diagnostics and to assist on and so as you're randomly going through the system you might trigger something that a diagnostic inside had been suspicious so that even now it doesn't crash, you get a message back saying this variable is not supposed to have this value at this time here's what's going on everywhere else that we think is relevant to look at - notice this. And the more diagnostics that you have, the more you're really looking at a what is usually called simulation-based testing. You run a simulator; you have a lot looking at the underlying variables and states and so forth and as the simulator runs you get to see what might be going wrong and you get to do some diagnosis. A simulator basically runs on its own. What changes on a constant basis is what diagnostics you're actually looking at you can't run all diagnostics on a real-time system because the system slows to a crawl. So you always only look at a small sample of the behaviors a small sample of the variables you could check.

On the other hand, you can add intelligence to your monkey instead of to the system under test. The diagnostics add intelligence to the system under test. The monkey, on the other hand, you can start saying "Hey, let me tell you about a statements sheet. If you're in this state and you get this sample, then it means you should probably get to this state here. And here's an indicator that you made to the next state -- check this variable in the what happens."

Some of the folks who do state based testing do it in a way that I don't think of as high-volume random testing. Instead, they use algorithms like the Chinese postman which basically tells you how many is the minimum number of steps to take you through one trip through the states. That's not high-volume automated testing.

High-volume automated testing is state based and you'd just keep it walking through the system for as long as the system has power. constantly checking: "What state should I get to from here? OK where did I get?" And you run that until you get tired of it or until you get a crash. But long before a crash you might start getting reports in the form, "Excuse me, I don't think that I got to the place I was supposed to get to. Here's the history of the states that I got to up to this point." And so it might take a long time before we start seeing some form of corruption, some delayed reaction that finally shows up as a functional failure to do the right thing. It's another form of high-volume test.

This is I think state of the art. I think this is going to be the general class approaches that are going to be used to certify critical systems in the future. A lot of this stuff is being done today. I don't think we have a theory associated with this stuff today. I think that company after company will compile stuff together. I think we're seeing tools that help people with parts of this problem and we are going to be reading more and more of this kind of stuff, better and better stuff over the next 10 years.

Slide 5.66: Applying Opposite Techniques to Boost Coverage

One way you can look at the distinction between exploratory testing and regression testing is to think of these as processes that have inputs and outputs. Why am I contrasting them? I think of these as extremes.

With regression testing the tester of course starts with the code to test but also with a bunch of different test cases that he will reuse and hopefully with analyses that show what those test cases were and how they were gotten and why somebody once thought they were worth something. The regression tester executes those tests, discovers some are out of date, some can be stricken, and generates two different types of documents. 1) bug reports and 2) improved tests with hopefully improve the analyses. The regression tester is focused on creating materials for reuse.

The exploratory tester, on the other hand, comes in with whatever information is available -- quite often models rather than test descriptions. Does testing, makes notes -- the testers notebook is a private notebook, mine not yours -- and from those scribbles writes bug reports that are fully reproducible, well troubleshot, perfectly reasoned, competent, analyzed bug reports. But the scribbles in the book are not conclusive, not inclusive, not full descriptions. They're not going anywhere outside this book. Two weeks later, a person comes back to try to figure out what to do next she might go back to her notes, she might have a review with her manager once a week or once a day

How does a manager collect status from exploratory testers? I'd go to their cube in the morning ask folks for status reports. It's kind of fun I had a huge cup I'd fill with coffee and go cube to cube and say, "What have you been up to, show me some of the best bugs you have found, tell me the strategy you're going with, the area that your testing how does it feel, what kind of risks are you encountering, what kind of support do you need?"

And that would be the current status report I would expect from exploratory testing. I'd get an idea of their thoroughness and their creativity. I might coach them on some tests that they're missing. I might encourage them to work in some other area and then find someone who could handle this area better. What I didn't expect was a lot of paper. It's not a typical output from this mode of testing.

Exploratory testing is not safe as the only approach to testing. There are some parts of the program that you are going to want to test in standard ways. Perhaps some of those are very well automated. But maybe their programs are changed over time. The exploration might give the background thinking for that sort of stuff but it doesn't take you there. Groups that do exploratory testing only tend to have plenty of surprises in the field. where something that was blind to every person in the group turns out to be an obvious mistake. You wonder how it could have been missed but it gets into the field. And yet the same groups brilliantly find some kinds of problems that following closely to the other techniques wouldn't have taken them to. It's one useful tool in a box, very useful but not the whole box.

Can we automate exploratory testing?

Now within the context of exploratory technique, what can we do to automate it? I don't know. I think teaching people exploratory modeling does more good than any note keeping system. The essence of exploratory testing is simultaneously learning about the product, learning about the product market, learning about the product risks, learning about how to test the product and what the test results have been, in order to design new tests. Anything you can do to capture that information in a way that can generate organized thinking about what the new tests would be is wonderful.

The ability to represent symbolically an essential part of the a system as you learn about it and then work that system and say, these are good pressure points for it, is a wonderful skill for a testers. So all of the stuff that you guys are doing with UML is relevant to skilled exploratory testing. You are really asking how do we teach people to think more efficiently about complex systems when you're asking how do we teach people to be better exploratory testers. Yes?

Module 6: Analyze Test Failures

Slide 6.2: Module 6 Objectives

So we’re still in the workflow test and evaluates. The last module was on test, time to evaluate. In particular our focus is on what you do when you evaluate and say the software under test is broken. The issues that I want to talk about involve investigating things that you think are problems, writing change requests, and trying to persuade people to actually fix things that think need fixing. How do you do that? I'm going to start out by making a point about persuasion.

The bug report is not a stand-alone thing. You do not write a bug report like you do something in your diary or your journal when you're exploring, right? It's not just something you write a little note to yourself and tuck in the back of your pocket and say, “Oh I found one. You know, I saw a purple spotted sparrow today how wonderful. Oh, and I found a bug too. I’ll put that on a different page.” The reason you're looking for these things maybe its aesthetic appreciation some people really think the variety of bug that they can find is just a wonderful thing, but *most of us are trying to find bugs in order to give them fixed.*

Slide 6.10: Championing Your Defect Reports

Our reports have value to the extent that we enable the company to make a sound business decision associated with the problem we found and to the extent that we don't help the company make that decision if we aren't giving the company value in our reports.

Slide 6.11: Discussion 6.1: What happens to your defect report?

Now I grew up in a sales culture and I learned at a very early, age 7, 8, basically as far back as I can remember, that there are two things that are involved in the sale. You motivate the buyer and then you overcome the objections. You make them want it and then you help them understand how they can have it when they say that I really couldn't have it. You close the sale when you have both things; they want it and they don't have any excuses for not getting it. We're selling bugs. We have to find a way to make them want to fix them and we have to find a way to deal with their excuses for not fixing them. That's the underlying thinking to my approach of Bug Advocacy.

So let's look at motivating first.

Slide 6.12: Motivating the Defect Fixer: Analyzing the Impact

It doesn't take very much to get somebody to think this is an important bug to fix if it crashes the program, erases the hard disk, and catches the monitor on fire. There are bugs that have caught monitors on fire - DOS 1, DOS 1.1, if you had a black and white monitor and a color monitor plugged in at the same time and the software didn't handle that correctly, poof - that was the end of your color monitor. Smoke was everywhere. It was a wonderful thing to see. People sometimes would put little Easter eggs in the program just to watch what would happen in the computer lab in ________________. Some people thought that was a bug. Some people thought it was a feature. In general something that dramatic doesn't take very much thinking. For something less dramatic you might want to find some way to describe this that would trigger some other response that is positive from the programmer. Sometimes that is literally just getting them to get intrigued by the problem. Gee, this is something that should work, I wonder why it doesn't? Sometimes, by the way, the written report is not enough. You have to walk up to their machine and say yeah, it's really strange, let me show you. Take a look at this. No that can't happen. Yeah, but it did. Look at this variation of it, hmmmmmm. As soon as you get that hmmmmmm, you are in good shape. For several minor bugs if you don't get that, it won't get fixed. If you can show that most of your customers will see it, even if it's trivial it's probably going to get fixed. A problem that comes up in install for every customer is often dismissed as oh it just happened once. But as soon as you multiply that by every customer you'll find somebody (probably somebody in marketing) who says every person experiences this in the first five minutes of using the program and you think it's minor? That gets attention. Another aspect that is very important to recognize with defect reporting is that you carry credibility. In fact a lot of your credibility in the company is based on the credibility that you've built for yourself in your defect report. How do most people see you? Do they see you as the person who produces the work that you produce? What kind of work do you produce that someone other than the other two testers and your manager that you work with see? Your test plans? Do you think that the marketing manager takes the test plans home at night and snuggles up and reads them in bed? Not likely. But a lot of folks take your bug reports home over the weekend toward the end of the project and asks themselves what is this? And if while they're reading at 2 AM on Sunday this thing is hard to understand, adversarial, cranky, and verbose they're probably going to remember your name and it won't be in the list of this guy should get a raise next time.

So you'll get reputation, positive and negative, based on these bugs with people that you might otherwise never meet or meet only once in awhile. And yet those folks might be in the room when your next promotion is the topic of an executive discussion. Your bug reports are your primary work product. It's worth thinking very carefully about how you present it.

So we have examples in the notes as well as these on some way that people who would fix the problem might be convinced that the problem is worth fixing.

Slide 6.13: Overcoming Objections: Think About Your Audience

On the other hand we have the objections. Along with wanting to fix a bug the programmer has to believe that it should be fixed that is to say there is not a reason for not fixing it. There are a lot of reasons to not fixed bugs, for example when you look at how many hours it might take to fix it and you look at how many hours are left in the schedule it might be impossible to fix it in the time available. It might be that fixing this thing is so likely to tinker with areas that you are afraid to touch, that you believe if you touch this, it will make more bugs that make this thing look like a mosquito where those look like flying rats – in Florida, we don't have flying rats we have enormous flying cockroaches though – palmetto bugs – which would you rather have, a flea or a palmetto bug? Well if you're worried about the plague the flea is bad but otherwise you don't want to risk unleashing the big ones. And that is a big reason for not making some fixes. Another issue that causes people to dismiss bugs, is that they just don't believe it is ever going to happen in the world. Nobody would ever do that they say as if they know. But if you report it in a way that makes it sound like nobody would ever do that, you're likely to convince somebody that no one would do that and then they're not going to fix it.

Right, these are objections to overcome but they are common objections. If you listed the top ten excuses for why we are not going to fix this bug, it would take too much work, no one would do it, it’s a feature – you know the list - and if you really want something fix and you know it's going to be subjected to that list, then you start thinking as you’re investigating that bug how can I come up with something that I can report or defend that will get past those objections.

Most of the time when someone says they can't reproduce the problem they mean I don’t understand your report well enough to reproduce the problem. Or they mean on the systems where I think they should replicate, it doesn't. Or they mean I did what you told me and I saw something but I don’t think it’s a problem here and I don't think you would have reported what it is I saw. All of those are fair statements if they are made honestly and in my experience most programmers make them honestly.

I've reviewed some very large collections of defect reports and asked the question how good is the communication in these reports and many, many of the bugs that ended up not getting fixed, started out as bugs that were hard to understand or that were really irritating. But especially hard to understand and if they were slightly hard to replicate they probably went by the wayside. But they went by the wayside because the person on the other side just basically didn't get it and in the interaction after that the tester would do stuff like say oh well if you can replicate it I can just close it then. Or the tester would say I have to pick my battles whatever. Or the tester would get offended and say this person never listens to me and would passively aggressively close the problem. Or the tester wouldn’t figure out how to say it a better way and so you see this long fruitless exchange that basically goes, I don't understand, well you should, well I don’t, you’re bad person. That doesn't get the bug fixed. It just makes for an interesting discussion on the next bug report and on they go.

Slide 6.15: Analyzing Failures with Follow-Up Testing

Now, there are ways to try to make your report more motivational before you put in. If you run a test and you get a small looking failure that does the mean that you have a small problem. When you run a test you have conditions, you have a defect in the code, and when the right conditions encounter the right effect the code, the right coding error, you get some kind of failure. The program does the wrong thing. And that failure has some visible symptoms. Now, if you see a symptom like it paints a little nothing on the screen it might be trivial and it might stay trivial but given one set of data it might paint something annoying on the screen but given another set of data it might take out the screen. Who knows? What you see is one little instance based on one set conditions. If you vary the conditions may be you'll find a failure that’s worse. And so the notion of follow-up testing to see if there's anything more serious is like seeing a little bit of ice in the water and asking, is that a little bit of ice or is it an iceberg? Maybe I should check.

There's another issue that you can look at and that is the issue of generality. If you're using the program and there's one thing that if you ever do this you'll see this failure, OK that's interesting. But suppose that I can get to this failure this way or this way or this way or this way, all paths lead to a broken room. That's a lot more nervous making than the one specific thing and you get to this bad place. Even if all 20 ways to get to this bad place don't show something that’s pathological, a maintenance programmer is likely to say it doesn't work correctly and everything passes through it. Either one of the paths we haven't noticed is going to be an unhappy path or I'm going to change something someday and what is benign today is going to turn out to be a tumor.

So if everything goes to a place that exposes a problem, that is more serious even if it’s a minor trivial problem, than if you can only get it on a rarely used path.

So when you try to convince someone that the problem is an interesting problem when you look at a minor bug and you say maybe there's more here, if I wanted someone to pay attention to this, what would I have to show? Well either there's a worse failure than the one that you saw or this little bitty failure comes up all the time if either of those it's true people will pay more attention to that report.

Slide 6.17: Analyzing Severity: Follow-Up Testing

So how do I do follow-up testing to look for the severity? Now I'm only doing this when I'm dealing with coding errors. There are errors in design. I do something and I look at the screen. It has a message I don't like. I will get the same message every time I get to the screen. It is the message that is intended by the programmer. I don't need to do follow-up testing for that. I need to write a note that says, I don't like this message and that explain why. But if I get a message that is of the form, only part of the screen was displayed and I know there is more there - that is the coding error, something is wrong. It is at that point I assume, here we are in a situation where the programmer has written code that assumes that the code that was just executed before this was working. If we have a mistake the program is now in a wrong state. The variables program is working with probably have the wrong values. And the timing to the extent that timing is assumed, it probably takes a different amount of time to do the wrong way into doing the right way.

So if we keep going with this thing, maybe there's going to be a little consequence. Because something made assumptions about the timing that is now wrong or the variables that are now wrong or about the state that is now wrong. The program is now in an impossible circumstance that is from the point of view of the programmer if you say can it get to here, they say of course not that would be a bug. And so everything they've done after that assumes something false - that the program didn't fail here. Given that the chance of discovering something else is pretty interesting.

Slide 6.18: Follow-Up: Vary Your Behavior

One of my all-time career favorite bugs happened when I was working for a word processing company that ended up commissioning the making of a board that would allow our word processor to run on an Apple II computer. We ran on the CPM operating system, which was a 8085 Z80-based operating system and Apple II was a 6502-based computer. But it was an open architecture machine so you take the peripheral board that had a Z80 processor on it, put it into your computer, and presto you now had an Osborne 1 equivalent computer with the Apple II keyboard. That general platform was actually for a while the best-selling platform for this word processor, a word processor called WordStar. And we decided to make our own board and I can't be project manager for that.

So, I had a board development group. I had an operating system group for porting the operating system so it could run on our board with our very fast Z80 what are lots and lots of RAM. And then we had to do the testing. I was the testing technology team leader for WordStar but I knew enough about the mechanics of the Apple II that they moved me over for this one project. And said, by the way manage this thing. The development is primarily done so you're mainly managing the testing the put on the project manager’s hat instead of the testing hat. We had an independent test lab for the testing.

They sent us a bug report one day. It was classed as minor severity, deferrable priority - low priority - and the bug was of this form: if you do this one thing it turned on the display, the display came up. Now I had of the belief that anything that was a coding error deserved a few minutes of follow-up testing just in case. If the program is in an impossible state, something bad might happen. And here we had something bad that was also fooling around with the devices.

I tried a few different things. The first thing that I tried was to basically do the same thing. Do the feature, see the light. Do the feature, see the light. Do the feature, see the light. Do the feature, see the light. Do the feature, see the light. That got boring pretty fast. Nothing bad happened. They knew it was minor and deferrable by the way because they would do the feature in see the display and then they would do a directory (catalog in those days) on the disk and say no, nothing got changed. Must have been minor, not a problem. Do the feature, see the light. Nothing bad happened. But it wasn’t supposed to hit the disk so I wasn't going to give up on follow-up testing just for that because I was just a little suspicious that may be it could get a bit worse.

The next thing I did was instead of changing my behavior for example my repetition, I started changing my behavior in terms of the next thing that I did was take a look at the feature that I was working with. Do this feature, see the light. Let’s try to do related features, do this other feature and see the light. Do this other feature and do my feature and see the light. That didn't work either.

Any feature that was related to the one I was testing, I could do that and come back, do my feature, see the light and nothing bad would happen. So doing tests that were related to what I was testing didn't matter.

But my next trick was to try tests that were related to the failure. Now I do the feature I see the light and then I do a disk related thing. Do the feature, see the light, take a directory. Do the feature, see the light turned out to be a failure case. A failure case? It erased the disk. I got to erase the operating system, the application, and all the customer’s data in one bug marked minor/deferrable. It's minor. Who would mind that?

I haven't always struck gold when I do follow up testing, sometimes a minor bug is just a minor bug. But you never know when you're looking at a minor bug how big it is. Is it just a little baby version of something that might grow up to be big? Like spiders, have you seen those tiny little spiders running around? Are those really tiny little baby spiders that will grow up to be not very big or are you looking at tiny little baby spiders that will grow up to be massive things that you don’t want to hear about so you better stomp them now.

You don't know without investigating further. I tend to spend at least ten minutes playing follow-up testing on any coding error that I see and I trust my instincts. Of course my instincts have developed over years, they're still not perfect. If you are a new tester, your instincts will develop over time but try ten minutes of varying your behavior. There those three levels of things that I mentioned try doing it over again, try doing something related to what you were doing, and then try doing something related to the failure.

If you don't see anything worth following up after that and you don't have any creative ideas let it go. But if you have a hunch that if you kept looking you’d find something, give yourself more time. People have asked me how long? It’s really hard to say. I've seen some tests be successful after over a day’s worth of follow-up testing. As you get to know the application under test, the operating system, and the programmer you're dealing with better, you may take the gamble that it is really worth taking this thing around for a day before you give up. And sometimes your gamble will be correct and sometimes you'll learn that this particular intuition is not to be listened to quite as closely. You build judgment by making mistakes but also by making successes.

Slide 6.19: Follow-Up: Vary Options and Settings

Another thing do you can do is to change the options in the program. Here we have program that is configured to do tricks and will keep 100K worth of data. That's great. And it turns out that while we’re testing we are tracing what's going on with the program and we fill up 100K right away.

So we say OK, may be we should have a MB worth of data, we can reset that. Maybe we should have 10 MB of data; we can reset that. The size of the database is an option in the program. A problem that might not look very serious if you have 100 MB available might look very serious if you only allocate 100K. And so if you see a problem that looks space related, maybe you're going to change the amount of space you allocated to squeeze it and see if it screams.

And then finally I might vary the data I'm working with. There are lots of program that operate from reference data into the extent that you have that, for example of reference data -- what language is your program operating in, like French or German - it reads a file, probably, that lists the names of the menu items and so forth, that's just data, of course we can think of that as settings too, but anything that it will read off the disk that will change its behavior in some way is something that we can say use this setting instead of that setting, I wonder it that will have an effect. You can spend way too long trying to change what’s going to have an effect, follow your instincts on what’s relevant. But ask the question, could this be relevant before writing the report.

Slide 6.20: Follow-Up: Vary the Configuration

Any the persistent setting that you can make in the program, any preference setting, might interact with the failure. Obviously you'll have theories of what might be relevant and you'll try those more likely than randomly resetting the settings. Another thing you can do is to change the hardware and software environment. How much memory is on your machine? How fast is your processor? What's the load in the background? All of that kind of stuff may interact with the problem and show that something is much worse than it first looked.

Let's imagine that you're testing on a machine that has tons of room, the latest patch of the operating system, really high-resolution video but on a card that has its own dedicated video RAM, no problem with the processor a really fast processor, and in that case you see something that seems to slow the system down and you say, gee, I wonder if it's because it's eating so much memory. And I wonder what would happen if I let it eat so much memory that it ran out. Maybe it's going to crash - out of memory. Or maybe something else will happen on the road to crashing.

So you go back to a machine that has high-resolution video but it's using main RAM instead of video RAM. Maybe you go to a machine that has the minimum amount of RAM allowed for your configuration. You go to a slower system so that you can see stuff happening a little better as it happens. In fact you go to the slowest processor that you're allowed to go to and still have it fit within the marketing department compatibility list. And let's suppose in that world you see a horrible failure.

How do you report this? I'd describe the failure, it corrupts data, it crashes, it does whatever really obvious thing it does, and then I'd put a note. By the way on a differently configured system the symptom is different. I get a performance hit but I don't get this other visible stuff. That note is going to help the programmer troubleshoot and it also meets my minimum standards for honesty in reporting.

I'm not being sneaky when I put my best foot forward which in this case means the most powerful failure forward. It was a matter of lock that I found the not so impressive failure before I found the impressive one. If I didn't testing on the minimum system configuration it would have come out the first time as really ugly. All I've done is switch from one legal configuration to another legal configuration and under the second configuration the thing was much more impressive so I report it as I found it on this configuration. I did, I found it on the other one too, but I found it on this configuration, here it is and by the way if you replicate it on this other configuration it doesn't happen the same way.

That's going to attract much more attention than "There's this minor little problem on this configuration but if you use another configuration it's horrible." The issue is some folks react when I talk about putting your best bug forward, they say, "You know that's not really playing fair. You're being too much of a salesman."

My task is to help the company make the best business decision that it can. And they can't make a good business decision if they don't have a clear idea of what is going to look like in the hands of someone who is in their target market. And just how bad it's going to look. And if you tell them on the minimum configuration it looks really awful and when you get off the minimum configuration it doesn't look so bad, they have the data they need but they will pay attention to it and read carefully because it looks awful somewhere and they need to know that. On the other hand if you say it does not look very bad most of the time but by the way there something really awful in this one case, they're not going to notice and they're not going to process it the same way and they may defer it just because they didn't realize how big of a problem it could be on that one system.

So as I change my configuration, each configuration that I test, I test as if this is the first one I'm testing.

Slide 6.21: Analyzing Generality: Configurations

On the generality side, the biggest generality problem I run into involves configurations. It is really annoying writing a bug report, having a problem that shows up clearly on your machine, it goes over to the programmer's machine which has way more RAM, an enormous hard disk, the fastest processor available on the market today, six other machine supporting this one and it doesn't happen on that system. And they say, well, it didn't happen to me. What's the problem, it's not reproducing.

I don't want to find this out two weeks from now when the programmer finally gets around to telling me that they can't reproduce the bug because there's a fair chance that after that amount of time, my machine has changed too. When I'm full time testing, I usually have one machine that just stays open and I'm swapping video cards and stuff like that all the time. Do I even know two weeks later exactly what the configuration was? Well, I tried to record it but some of the things I swap out I never expected to swap out two weeks ago.

I don't just test on one machine. It's common for me to test on two machines, for example, at this point I have a fairly recent vintage Dell and I have a Micron machine that is a P1, 120MHz, a lot less RAM, a relatively small hard disk – 8 GB, and a very different video, very different keyboard, it's got the IntelliMouse keyboard driver that is incompatible with some programs and if I replicate, if I find a bug on this one, that's great. This now turns into my bug reporting machine.

The next thing that I'm going to do is swing over and say, okay, let me replicate it on this guy. On this machine, I now have a bug report form and can type stuff in. now on this machine I say okay the first thing I did with this so I'll do that. Okay. Do this, got to do that. Step 2, do this – yeah I did that. Step 3, do this and you should see a message – what's the message? Oh, yeah, here they are character for character exactly the same message, exactly the same capitalization I can see it right there so I can get it right. Different strings have different meanings. If the programmer searches for the string you typed in and can't find it, that becomes a "cannot reproduce" because of your sloppy typing.

So do this third thing, and you'll see exactly this. You want checkpoints, if you have long sequences especially you'll want to have times where you say, BTW this is what it should look like now on the screen. So finally, Step 5, you say, do this and this happens. And if there's any doubt that this is a bad thing, you say, I was expecting this other thing to happen.

So you have Step 5, one of two things happen. Either you get the problem, if you get the problem on my two machines, they have one processor – a different processor, one version of the operating system – a different version of the operating system, one video driver – a different video driver, one keyboard driver – a different keyboard driver, a different mouse driver. There's basically nothing in common between the two machines. No version of anything is in common. No manufacturer of any part is in common except that Intel makes both chips and that's a coincidence – if I was really doing a lot of special testing at this point I'd probably have an AMD in one of these just so I'd have yet another thing to be different. And if those two systems replicate the problem, I don't have to worry about it not replicating on the programmer's machine. And if it does fail to replicate on the programmer's machine, I can find another machine right off the bat. Okay, it doesn't work here, let's try it over here. Pretty quick.

On the other hand, if it does replicate, if it turns out, it's not the same on this machine as that, I know before I write the report. I can do one of two things.

1. If the failure is pretty serious, especially if we're close to the end of the project the company may have a policy that says as soon as you can make a serious failure reproducible you must put it into the bug tracking system. That is not an uncommon or an unreasonable policy.
2. Another variation of that is that any bug of severity X or higher (some people have severities of 1 to 10, some people have them from 1 to 3) any severity X or higher must go into the database before you go home at night. Late in the project you don't want to have late surprises, you want to get the worst news in as fast as possible.

Well, if you have two machines that differently configured and you can replicate on one but not the other, it might take a while to find out what the critical component is, so you're going to write a report that says "bad thing" as your title, then describe the steps you went through, and actually it says "bad thing" "configuration dependent" you go through the steps, you say on the machine where it failed, this is the failure, these are the details of the machine where it fails, these are the details of the machine where it doesn't fail, more to follow. And that's what you report if you have to report it tonight before you've done the troubleshooting.

Now, that might be plenty for the programmer to understand a few things to vary, check some code, what could possibly be different that might be triggered by configuration difference, find the problem then come back and say, is this the replicating machine after all you might get that in the morning. Or the next morning you might go through and try to swap back and forth to find out what’s not working.

It’s way better to know that there’s a configuration difference in advance and report that as part of the characteristic of the defect than have someone come back to you and say, I can’t reproduce it and then you discover together it’s a configuration problem. In one case it looks like you’re in control and in the other case it looks like you don’t know what you’re doing. For your personal career benefit, the more you look like you’re in control, the more you get to think in terms of raises and happy times. Add value.

Slide 6.22: Analyzing Failure Conditions

Now, let’s come back to the issue of objections. We started out saying that there are things that make people want to fix it.

So, we looked at ways to get it to be as secure as we can. And then there are reasons the programmer might say, yeah, yeah, it’s a bug, it’s significant but you know there are reasons why a decent person wouldn’t do this right now. And first on this list involves unrealistic, perceived unrealistic problems. Suppose that you are testing the field 20-50 and if you enter something that has 65535 characters into that field the program crashes. And you write that as a bug report. Odds are the programmer is going to write a note back that says, don’t do that. Not a bug. That’s a not very credible report. It might be more interesting to write a report that says, if I give it something that is 3-characters or more, it will crash. It accepts 3-characters and then fails. If that’s true.

Slide 6.23: Uncorner the Corner Case

Just because you found it, the reason you take an extreme value is because that's your fastest road to a failure, we're looking for the best representative. The best representative gives us a slight edge over the rest of the equivalence class for finding a defect, for seeing a failure. Once we see the failure, we see something that might be because of the slight edge, the extreme case, or might be applicable to the entire class.

So if you put in 65000 characters and it fails, that might be because it's 65000 or it might be because it's more than 2. Don't report 65000 until you check 3. Because if you write a report that says any number bigger than 100, or bigger than 99 that causes a system crash, you're not going to get a response back that says, well no one would ever do that, that's ridiculous. On the other hand, let's suppose that you really have to be pushing 65535 to get a failure. Try 65534, no problem, 65533, no problem, 100,000 no problem, 10,000 no problem, 65,535 – crash. Okay, now your bug report reads, I have an odd thing at 65535 characters. I have this 2-character field that I'm supposed something into and if I paste 65535 1's into that field, the system will crash. I tried it with 65534 and it didn't crash. I also tried it with these other guys.

There's a pretty likely case here right, people will read that report and say, thank you so much for sharing – closed. On to the next one. They might investigate a little more thoroughly. They might have a policy that says no crash bugs will go unfixed. But, they're not going to spend a lot of time on this because it is an extreme, unusual case. If our task is to help the company make the right business decisions, then having the executives know that this really is a most extreme case, suppose that you have 300 bugs that need to be fixed and enough programmer time to get 200 of them fixed; is this one of 200 that gets fixed or is it one of 100 that you want to have just stay? Maybe you'd like to make sure that the right 100 are chosen to survive in your program. When you give people information on the limits of a failure, you help them make that decision between does this belong in the 100 that won't get fixed or in the 200 we have time left to fix? The time we spend fixing this one will mean some other ones will get bumped off the list and into the 100.

So knowing that it is narrow, and it really is an unrealistic case is valuable if you demonstrate that it actually is. People will appreciate that. Great troubleshooting, we can defer this one with confidence. Or they might say we don't care that it's narrow, we're going to fix it anyway. Thanks for letting us know. The critical point to recognize is the condition under which you found the problem is merely the first part of your analysis. Any condition under which you can replicate the problem is a fair condition to report it under.

Slide 6.24: Analyzing Non-Reproducible Errors

Our next issue is the issue of non-reproducibility. This is probably the most common excuse for not fixing a bug. I call it an excuse because many programming teams won't even look at a bug you admit is non-reproducible. Because of that, many test organizations have a rule that if you can't reproduce it, it can't go into the system. That's a horrible mistake, terrible mistake.

Simple example. I was a programmer in a telephone system for a while. One day we came in and the system had crashed, dead. Phone systems don't do this. Phone systems are fault tolerant. Phone system stay up for 40 years. Our phone system had crashed, dead. This was bad. We checked everything the system had saved to the disk. We checked the event log that had gone out to the printer. Something had happened before midnight, actually before about 10:00. At least the last event we saw had happened about 10:00. That's all the information we got. We wrote down everything that we could find, but basically we had a no power system with no lights on and nothing going on. Didn't work. Thanks, that's useful.

So, we hoped that elves had come and turned off the machine, who knows? Nothing showed up for quite a while and then one day we came in and we had a dead system again. But we wrote everything down and said, another dead system, and got more nervous and crossed our fingers and nothing happened for a while. And then we came in, and we noticed that the first dead system was February 1, the second dead system was March 1, and the third dead system was April 1, and now we understood why we had delays between the systems and we also noticed in terms of the times of the logs anything after midnight on those days.

So we said, there is some end of month processing going on in this system. We didn't know what it is, but we now had it narrowed to about a 15-minute time window. And from there we were able to find the bug with a lot of annoyance but we zeroed in and discovered exactly what the problem was.

How could we have found this without tracking every instance of the failure and putting it into the system even though it was completely non- reproducible? Instead we would have shipped the thing and said it was non-reproducible and one day, the start of the first month that it was in the field would have called us on their cell phone and said, it doesn't work. My system is dead. Of course, they wouldn't have reached us because our system was dead too. But we would have dismissed it as not reproducible, it doesn't matter, you don't put that in the bug tracking system.

You have to put it in otherwise you will never be able to identify the patterns. Of course you have to be able to treat your bug tracking metrics in a sensible, mature way. And some companies are incapable of this. There has to be a state in which the bug is left open for the purpose of finding patterns and yet is considered closed/non-reproducible or closed/duplicate for the purpose of counting how many active things are left in the system. But you want to have it open for the purpose of rummaging through and every Monday morning come in and say I wonder if, let me pick this failure, how many things are like this one, let's take a look through.

At one company I was at, we called this suspended state, the dumpster state, take these bugs and throw them in the dumpster away from the corporate quality control people who counted metrics so we put it in a state that they couldn't actually see. And then every Monday we did dumpster diving. Let's sample some bugs, take a look and see if we can finally get 5 to 10 bugs that compare to this. When we get a population of bugs that appear to have similar system, pull them up, make them active, assign a programmer team to it and see if now we have enough information we can find a common pattern and break the bug. Sometimes we could, sometimes we couldn't, but for non-reproducible bugs that you take seriously that's an important thing to do.

There's another reason why you want to report every non-reproducible bug that you can find as long as you can put some data with it and that is that maintenance programmers who are serious about this can often find problems, make them reproducible that you can't. After all, they have source code, they have trace, they have debug. They have the memory of how they coded this thing. If you tell them I was just happily doing something kind of like this and I got this particular bad message, they know there are only three ways to get that bad message so they work backwards from there.

When I was doing maintenance programming, 20% of the bugs that came to me as non-reproducible, I could fix without further input from the test group. I've talked with other folks who consider themselves good maintenance programmers, I was an okay maintenance programmer, and they would remind me that I was an okay maintenance programmer, they would say, oh, 20% well you're trying, that's nice. Good boy.

An estimate I got from someone who was an experienced maintenance programmer was 80%. Now they assigned programmer teams that worked like mad for their bugs but for a class of bugs that were non-reproducible that they said, this is on the list of ones they want to check out and that check out was on severity, not based on how promising it was, they were able to break out 80% once it had gotten to the programmers and get them fixed. That’s the largest percentage that I’ve heard.

And then, I get the folks who say, well, if it’s not reproducible there’s no point even looking at it, you can’t fix it. Zero means we’re not trying. If you’re in a company where they’re really not going to try, it still makes sense to report the bug into the tracking system so when the problem shows up in the field and people say, gee, who’d ever imagined that this could happen, you could say, well, actually, I could and in fact, I did. I tried to tell you about this and then you can have a discussion about troubleshooting standards. Always report the problem. On the other hand, if you think that you don’t know how to reproduce it, try. It is better to make it reproducible than to report it as non-reproducible.

Slide 6.25: Analyzing Non-Reproducible Errors

A test case is a very complex thing. We have things we control intentionally and things we don’t control intentionally. When you’re running a test you specify certain values but you don’t specify others, they just happen. For example, there was a period when I was consulting at Hewlett Packard, we had a new version of printer on the floor and we would put tests in this printer and it started giving us really wild and crazy results. We got very nervous, I was with the group doing firmware testing, we got very nervous about the code.

Here’s the actual problem. There was a fan out here. The printer wasn’t getting any cold air blowing on it. It was a prototype board; it ran very hot. The memory was overheating. The random behavior we were getting was hardware. Did we control the temperature on this printer – not consciously. Was the temperature on the printer varying to a range that was affecting the behavior of the software – yeah. Stuff happens that is not in your control that you’d never thought of as the reason you’d have a failure.

So when you see a failure, there are the things you are paying attention to, and then there’s everything else. When you have a non-reproducible failure, one of the critical conditions that caused the failure is in the population of “everything else” and not in the population of things you normally pay attention to.

Recognizing that allows you to go to the next step. Saying, okay, what’s my second tier, what do I normally not pay attention to, maybe I should start thinking about that now. And it’s worth building a list of second and third tier conditions to say, well what was the state of this machine when we started this test. In the notes, I give quite a few of those second tier conditions. Yeah, several pages of them.

Slide 6.26: Analyzing Non-Reproducible Errors

Let me give a few examples here. Timing is a common thing people ignore. If you have a stack overflow it didn't happen on the last case. The last case is simply the last case. But the stack got full over time. If you have a memory leak and finally your program crashes out of memory, it didn't leak all that memory on your last test case. Now here's a classic failure to reproduce situation. You test printing and every time you print there's a leak.

So you print, and print, and print, and print. You print 200 documents one at a time, you finish testing printing, you haven't rebooted the thing, now you bring up the program and you draw a circle, it crashes - out of memory. You go, ooo something is wrong with the circle. No. What was wrong was the printing leaked so much memory that there was 2 bytes free and the next feature that was going to come along was going to get hijacked and crashed. But that's not how you perceive it. You reboot the program, you bring it up, you say – a circle that's really bad. You draw a circle and it doesn't fail. You say maybe it was a little bigger circle. You draw a little bigger circle and it still doesn't fail. Was I using a different color? Maybe I was just over a little further to the side of the screen. Gee, what happened? You are not paying attention to the sequence of things you did before, right?

As soon as you realize that you can't replicate based on the current data, you say, what was I doing before? Well, gee, I was print testing but I did tons of that and there was no problem. Maybe I should tons of that again, there might be a problem, let me check or let me turn on memory monitoring and see what happens if I print a few times. And now you find a problem. If you don't think backwards, you don't see it. And if thinking backwards isn't on your list of how to think when the things you pay attention to aren't the things that turn out to be critical cases, you're never going to notice that problem.

So time delay failures in my experience are the most commonly not thought of failures that testers who fail to reproduce could think of and do something about. We have lots of other examples.

Slide 6.32: Writing the Defect Report: Make It Clear

A key issue involves the clarity of the reports. It was fascinating doing a study particular for one company that had very high quality standards but it had some recall problems based on problems that had actually been found but dismissed. And so I looked through several generations of products reports and it got to the point that I could predict what the resolution would be: defer, don't defer, not a bug, and so forth, by reading the report for style. A report that was too hard to understand was almost certainly not going to get fixed. A report that was really simple and clear and easy to understand but really minor got fixed anyway, not a problem.

A simple lesson from this: you got a problem you want to get fixed, make it look simple and easy to deal with. You want it to not get fixed, bury it in a description of what you did on your summer vacation. Oh, you know I came in last Wednesday and I did this and I was thinking about that – it's amazing what personal diaries get tossed into bug reports sometimes, don't do that! I was thinking about this or I was doing this test and I had this idea for another thing and you know I was thinking about the low quality standards of this person who might have contributed to the code and it's really unfortunate that you people won't fix bugs like this so when I saw something that kind of looked like that I went off and did these other things too to see if I could come up with something more powerful so you wouldn't ignore it . . .

You've seen bug reports like this, right? They don't get fixed. They bury the critical information in chat. If you really have to say these things to the programmer, send her a love letter separately. This is in reference to how you're going to handle bug 402. I know you won't like it but here's why you should think of it. But in bug 402 itself, be crisp.

The smallest number of steps in short, easy, simple sentences. The simplicity of the writing is extremely important. Many, many of these bugs are going to be fixed late at night, just before the release by people who are exhausted. I want you to think back to when you were college, you were studying, and you didn't have the sense to go to bed at a reasonable hour, and so you'd be studying at 4:00 in the morning and you'd be reading some paragraph and you'd get about halfway through the paragraph and you'd realize you didn't know what you just read.

So back to the start of the paragraph and you'd keep reading and you'd get to the point where you didn't know what you'd read.

So back to the start of the paragraph. . . You'd do this for about an hour because you're too tired to realize you're into this loop and finally you realize, I have to stop processing. You grab a cup of coffee, you get up and walk around, you go to bed, you do whatever you do but you give up on that paragraph, it's done. It didn't happen with the other paragraphs, what was special about that one?

Think about writing style. It might too many transformations. Let me give you an example of a transformation. The boy hit the ball – is a simple sentence. The boy did not hit the ball – is one transformation away from the simple sentence. The ball was hit by the boy – is another transformation away from the simple sentence. And there are people who have reading disabilities who can handle one transformation sentences. But – the ball was not hit by the boy – whew, gone. At 4:00 in the morning when you're too tired, your processing capacity is not 100%. The further away you are from simple sentences, the less likely it is that you can figure out what the simple sentence underlying this thing was and the more likely you'll have to go, uh, did not compute. Let's go back to the start.

So here we have a few complex sentences concatenated into one really long one with the punch line kind of ¾ the way through the second half of the sentence. You're not going to get there if you're ¾ asleep and neither will the programmer. This report is going to turn into a paper airplane. Goodbye. Don't do that.

Short, simple, declarative sentences, name every step, don't skip any, start at a place everyone knows how to get to, number your steps, make separate paragraphs. Step 1 do this, Step 2 do this, Step 3 do this. If there's any ambiguity at someone may be seeing at this point, just put a little checkpoint here, by the way, this is what the screen should look like, by the way you should have just heard a beep, and on we go until finally we hit the critical step, Step 6 do this and this happens and if there's any question about what the problem is, you end up saying I would have expected this other thing instead. And they can go, oh, yeah.

Now it's not just exhausted programmers who have this problem. If you are a development organization that spreads your development across five continents, you don't think that somebody that normally writes in Russian is going to write their bug reports in a way that will be easily understood by somebody who normally reads in Japanese. The English that has to be written for a multilingual development scheme to fully understand what’s going on, has to be fairly simple. Now, even if you’re not writing across continents, even if you’re just writing in the US, how many people on your programming team were born in the US?

The simpler you write, the more likely it is that you will successfully communicate with the person who’s been tasked with trying to figure out what you’re saying so he can try to fix the bug. If you want to sabotage the process, write in flowery sentences. This is not creative writing. This is simple, functional, technical writing. Get it clean and simple.

Slide 6.33: Writing the Report: Keep it Simple

Simplicity is another key piece. If you report four related bugs on one form, one will get fixed, the bug report will get marked fixed, and the other three – well, maybe you were lucky and they got fixed too if they're related, maybe they turn out to be duplicates and maybe they don't.

My rule of thumb is pretty simple, if there are two different tests I would have to run to see whether the bug has been fixed, it must be two bugs in terms of what I'd report. If I'd have to do two tests to replicate the check, then I'd write up the two forms. Now in some companies, there are so many politics associated with bug counting metrics that you do what you have to do to deal with your quality control. I can't talk about dysfunctional quality control and how to manage those people. But if you're just looking at good engineering, if you different symptoms or different steps, write them on different pages and cross reference them.

Sample test files are probably the most controversial piece of advice that I give. My suggestion is that every bug report should be self-containing and the sample test file that comes with it should be a supplement that isn't necessary to reproduce the bug or understand the bug. Other folks end up saying, wait a minute, this is probably an executable, they can just run it and see the problem. Where's the screen shot, they can look. And that's wonderful. It's great to have that extra stuff until the system administrator decides to purge stuff off the hard disk one day and there go all your supplementary bugs – well, nobody has used them for six months category. Yeah, you can restore them from the backups when you finally realize two years later that this thing is there. If you want to preserve the ability to reproduce the bug, preserve it inside the report.

Another piece is you can't necessarily assume that the programmer has your test tool or is willing to use your test tool. And so you have a test execution tool that saves things in its own format and will execute those, so you send the script and say, just run this and you'll see. It's not a safe bet that that programmer will actually run it. You just can't assume that programmers will work with your test tools. Not unless there's a corporate directive and incentives and so forth. Some companies operate differently than others, but it's very commonly the case, that anything that you put in the context of something that is proprietary to the test group will not be considered data by anybody else. It will just be considered a null attachment that takes up space.

So put it in plain text if you can inside the bug report.

Slide 6.34: Writing the Defect Report

There are several aspects to a well-written report besides the simplicity. If you can reproduce it, explain that. You keep it short, as short as you can. If you have a bug report from the first time you found the bug and it was in a sequence of 30 things, you know if you got trace tools, you're wondering through playing with the program, and you discover that between this starting step and this starting step you got a failure, does that mean that those sixty steps are all the ones that were needed? No. It might just be three.

So start here, okay, and then somehow skip 20 of them, get rid of a bunch and see if you can still get to the failure. If you can, keep throwing steps out. Eventually, you will find one that if you throw that one out you won't see the problem – ooo critical step. Get rid of some of the others around it. And eventually you may get to something that is maybe five steps, maybe more. But where every step there if you don't do that one you're not going to see the failure.

That's the report you want to write. Minimize it. The longer the report the more somebody is going to say, no one would do that. This is a special case. The shorter report the more somebody is going to say is that could happen. Boy, all you'd have to do is breathe and it fell over. But don't skip steps. Assume this person is going to read literally and just type what you said. Do this, okay. They're not going to do this exactly the way that you said, that's why in your report, do this – check – test that it will work before you put it in.

Don't call the programmer an idiot in the bug report. It makes them angry. It makes their manager angry. It makes their manager talk to your manager about how you're beating up their programmers and that causes them to be demoralized which causes them to less productive. And that makes your manager say to you, I need to read every bug report you ever write so that we can edit them so that you don't write these nasty bug reports. And that takes your productivity and cuts it to a third.

Now that doesn't mean that you shouldn't state facts as facts. You don't have to talk around the fact that the program crapped. Some people get so intimidated that they might offend someone that they end up saying, the program behaved in an anomalous way. The program didn't behave in an anomalous way, it printed the wrong number. But "printed the wrong number" is a different statement from "it printed a really stupid number". Or "it printed 12 when I expected 6" is a clean, factual statement. Be direct but don't write in a way that a reasonable reader would believe you are being a jerk.

Slide 6.38: Writing the Report: The Headline

A headline is a special field some people call a summary or a title. It's the short one line description of the problem. And the short one-line description of the problem is very often the only description that people will see. Summary reports of all the open bugs for example are often printed as bug number, short description, severity. And you just go through, here are the severity 1's, the severity 2's, and so forth. People coming into triage meetings may only read the summary lines. Executives who skim the list of deferred bugs will say is there anything that will cause me to intervene –probably will only skim the summaries.

So if you want your problem to have the most impact it can have, it has to have that impact on the summary line. The summary line has to be the invitation that says, read more. Now there are a few important aspects of the summary line if you want to achieve that. One is that your summaries always have to be credible. Don't write as a summary, horrible, terrible, really bad, serious bug, and then inside, I really hate these missing commas. That's the last really bad, horrible, serious report anybody will read that has your name on it. So the rule is: *no exaggeration*.

Another important piece about a headline is that many people will only read the first few paragraphs or first few words. In fact, nobody is going to read more than 70 characters depending on how the final report is formatted. I know that in many systems you can type in 255 characters or even more, but the way the reports are printed, you get about 60-70 characters and it just cuts it off. And so if you have a headline that you come in and start describing the thing, "Oh, yes, there's this really interesting problem that …"– and then you start talking about the feature but it's gone.

They're not going to see it. Even if all of your text appears in the first 60 characters, the attention of someone who is skimming this list asking if they need to pay more attention to it is limited. If they look at the first three words and they're boring, they're gone. They go to the next thing.

People who write for newspapers understand the Important Rule; the most important words go first. *You never know when your reader is going to stop reading.* Everything you've written so far should be more important than anything you have left. Same thing with the headline, you want to capture what the failure was in a way that the person who reads this can understand without having to be an expert in how to use the program.

If executives read anything, they will be reading a report somewhere of these are the outstanding problems in the system and they're only going to be reading the summary lines. And they're going to read them fast. And they're not going to read them thoroughly. You describe the problem quickly, to a degree that a reasonable ready can visualize the problem more or less accurately. And given that they can now imagine what the failure is, you want to give them some information to help them understand how serious it is.

Now when I say, help them understand how serious it is, this isn't the place to say, serious, minor, you've said that in another field. This is the place to talk about consequence briefly, in 60-70 characters. But in the ideal case, you are reporting what went wrong, why they should care, and any constraints that are on this thing that are so important that it would be dishonest to write this report without telling somebody that it doesn't happen under this circumstance or it only happens in this event.

So those are my three kinds of content for a headline. Every bug's headline will be read. Many bugs will be read only if the headline is a good enough teaser to get people to say I need to know more about this and I think I need to look at this.

Slide 6.44: Exercise 6.3: Defect Reporting (1/18)

So with that in mind, we're going to look at a bug and we're going to write headlines.

Now a more complete version of this exercise will take longer, substantially longer, and you could write the entire report. And my feedback on the exercise will tell you what it's like to deal with the entire report. There's a fair bit of description of this exercise in the notes as well.

This is based on Windows 95 Paint. Windows 2000 Paint has the same bug. I haven't tried it with Windows XP but I imagine it's still there. As the tester in the class, I would urge you, if you are a student in the class, please don't spend time replicating the bug, but as the instructor replicate the bug and play with the conditions before you come to class. It's going to be really valuable being able to say, I tried this and I tried that.

Here's our program.

We open up Windows 95 Paint. Some of your students will not know the difference between a Paint program and a Draw program.

So it's useful to point that difference up right away. In a Draw program you have objects, like circles. If you draw a circle, there's a thing called a circle. You can grab the circle. You can move the circle. You can delete the circle and everything else stays the same. In a Paint program, the circle is dots. All you're doing is taking a paintbrush and going bump, bump, bump, bump, bump, bump. If you want to move the circle, you have to take the thing that the circle is painted on and move it. You have to move all the dots that are in the circle because there is no circle, it's just black paint. Now, because we're going to be doing cutting and pasting and moving, it's going to be valuable for us to be able to see exactly what it is we cut and paste and move.

Slide 6.45: Exercise 6.3: Defect Reporting (2/18)

The very first thing we’re going to do is select the paint can and color the background a different color from the system white background. We’re going to put a layer of paint on top. Then when we do a cut, the gray is going to get cut and so we’ll see exactly what we’re doing every time.

Slide 6.46: Exercise 6.3: Defect Reporting (3/18)

Let’s color the background gray. There it is, it’s gray. And now we’re going to start looking at selection.

Slide 6.47: Exercise 6.3: Defect Reporting (4/18)

We want to cut something, first we have to select it. There are two different selection tools. The first selection tool is the Star. And with the Star you can do what’s called freehand tracing. You draw any shape and what’s inside that shape gets selected and can now be cut, moved, deleted, or whatever.

Slide 6.48: Exercise 6.3: Defect Reporting (5/18)

We’re going to draw a circle, more or less like this, and the only problem is what we’re going to see is the bounding box. This is not a bug, well, it’s not an unintentional bug, this is how they did it. In more expensive Paint programs if I was to draw a shape with the freehand select tool, what you would see is the selection area would be the exact shape that was selected by the freehand select tool.

In this, you got what you paid for, free program that comes with the operating system, they avoid this very rich source of bugs. I’ve worked as a test manager on projects involving several Paint programs. It is remarkable in how many different ways you can have failures trying to draw a marquee, the little line of moving dots around a selected area – how many ways that cannot work. And perhaps for that reason, the engineers at Microsoft said, oh to heck with it, it’s free. They can draw their circle but what we’re going to draw is the smallest box around the circle that will contain the circle, the selected area. So that’s what we see.

When you write up the bug report, this is not a bug, this is a feature.

Slide 6.49: Exercise 6.3: Defect Reporting (6/18)

Let’s go out of the program and come back in and color the background gray and do our next test. We draw a circle. We can draw the circle using the circle selection tool. Hello, circle selection tool. And we’ll draw a nice little circle and then we’ll go to the freehand selection tool and select around the circle and cut and it goes away. And so what you saw here was that it looked like a nice perfect square area that was cut but actually it was a pretty jaggy area that I’d selected. The square around it was just the smallest square that would fit around the area I selected. But the circle is gone.

So that worked.

Slide 6.50: Exercise 6.3: Defect Reporting (7/18)

Now I’ll go out of the program and come back in and color the background gray and draw my circle and select around my circle but instead of cutting it, I’ll drag it. And there we are, it dragged. You can see that, there is white in the background where there used to be gray. By setting this thing up with the gray background you can see the effects of the test. It is important to structure any test you run so you can see all the effects of the test, or at least lots of effects of the test, you can see what you’re manipulating and not just what you want to be manipulating.

So here we can see that we’ve moved the right stuff. Oh, goodie. Freehand selection seems to work.

Slide 6.51: Exercise 6.3: Defect Reporting (8/18)

We’ll go out of the program and come back in and color our background gray and draw our circle and now use the Square selection tool. And with the Square selection tool you draw your nice circular area and it selects the smallest square around that but it fully selects that and it says, oh you wanted a square, so now we’ll drag and we’ll see that a perfectly tidy square area gets moved.

So that works too.

Slide 6.52: Exercise 6.3: Defect Reporting (9/18)

This is boring. This is boring because we're not finding any bugs. The program works too well. When the program works to well, you need to up the harshness of the tests. The only reason you run simple tests, is because you think the program wouldn't pass simple tests.

As soon as it passes simple tests, light the fire a little hotter and do a combination test. So now, we'll still do cutting and pasting but let's do cutting and pasting under circumstances the programmer didn't bother doing all the time. We'll zoom to 200%. We could set zoom to 800%, whatever, but let's set zoom somewhere away from the default.

Slide 6.53: Exercise 6.3: Defect Reporting (10/18)

When I say zoom it, what I mean is at 200% we’ll stretch the canvas that you’re painting on so what used to be one pixel wide is now two pixels wide.

There’s the zoom box.

Slide 6.54: Exercise 6.3: Defect Reporting (11/18)

Here we have the same size window but because we zoomed by 200% what was in bottom right hand corner now occupies the entire window that we can see. If we want to see more, that’s no problem. We just click here and scroll up or click here and scroll to the left. And so we could see any part of the original screen that we wanted. But what was this has now been expanded so that made twice as tall and twice as wide, won’t fit inside the original box only ¼ of it will be visible and that’s our ¼. Great.

Slide 6.55: Exercise 6.3: Defect Reporting (12/18)

Now let’s select part of that visible circle. We use our freehand selection tool, draw kind of a square, circle-ish, jaggy whatever and do CTRL-X. And what happens is nothing. More precisely, what happens is the marquee, the selection dots around it, they go away. The circle stays. The selection marker disappears. Time to do a little follow up testing.

Slide 6.56: Exercise 6.3: Defect Reporting (13/18)

We go out of the program and come back in, we'll color the background gray, we'll draw a nice circle, we set a 200% zoom and we'll select the area and we'll do a move this time instead of a cut. It moved. If you cut, it doesn't. If you move, it moves.

Slide 6.57: Exercise 6.3: Defect Reporting (14/18)

We go out of the program and come back in, we’ll color the background gray, we’ll draw a nice circle, we set a 200% zoom and we’ll select the area and then we’ll move a little bit and then we’ll cut. In that case, it moves and it cuts.

So cut doesn’t cut unless you move first. Something is wrong with the initialization of cut or something like that but in the narrow case, cut first doesn’t cut it.

Slide 6.58: Exercise 6.3: Defect Reporting (15/18)

So with just a little more follow up testing, we come out of the program and we come back in, we color the background gray, we draw a nice circle, we set a 200% zoom, but I want to see more of what’s going on so I expand the window.

Slide 6.59: Exercise 6.3: Defect Reporting (16/18)

I grow the window, we’re still at 200% zoom, but the window has grown bigger so we can see the whole thing and now we’ll select the lower right hand corner. You can see the selection marker, there’s a box around the circle and we cut that but instead of cutting all of this, …

Slide 6.60: Exercise 6.3: Defect Reporting (17/18)

… this funny little area here gets cut.

Slide 6.61: Exercise 6.3: Defect Reporting (18/18)

Time to write a bug report. By the way, this bug is fully reproducible. See there it is again. Alright, in fact if you did it on your machine at least up to Windows 2000, I just don’t know with Windows XP, but at least up to Windows 2000 you would have the problem. Now the bug report assignment is either:

- just write the headline, or
- write the headline and write the description of how to reproduce.

For this group, you’re just going to write the problem summary. Take five minutes and write the problem summary. Have a nice time.

Slide 6.61: Exercise 6.3: Defect Reporting

So everybody I think is done writing the headlines. I want to make one suggestion for the folks running the exercise in their class. Make sure that the students work simply from the printed page. Don't have them actually reproducing the bug on the screen. You are trying to get things that can be compared. And as they do troubleshooting further, they are going to find other interesting things about the bug that will affect how they write everything up.

What I'd like to get from you are some headlines. Now some of you might be a little shy and might not be willing to have anything you write appraised by other people. After all, how do I put this, as a tester, most of your writing is going to be appraised by other people whether you like it or not.

So you may as well put it up here and have it appraised by friends. But is anybody willing to have their headline come up here so we can all look at it. I will make some comments about it but they will tend to be helpful. Okay.

- Cut fails when view is zoomed 200%.
- Cut fails at 200% zoom.
- Cut doesn't work on all versions.
- Cut operation malfunctions
- Paint 95: at least two different cut function failures at zoom 200%.

Okay. I generally get a little more diversity. I'm not sure if that's my teaching or if that's the Unified Rational approach. But I agree with you, there are two different problems that I would write up in two separate reports.

Let's look at how to analyze this thing. First I want to look at the observed failure. There were two different failures.

1. It didn't cut.
2. It cut the wrong thing.

If I see two different failures, that's two different bug reports. Now let's take a look at what happens when you try to combine two different bug reports under one bug. Let's keep it simple, right. Just describe one, after all they're probably related. It says cut fails. Now if you read all of the headlines in the bug tracking system and translated them all down to their essence, it would be "feature name doesn't work". In this case, "feature name" is "cut". But, Feature name doesn't work when zoomed 200%. Feature name doesn't work at 200% zoom. Does that tell you anything about the failure? This says something's wrong with cut. If all of the bug reports in the tracking system were of the form "something is wrong with feature name", how many of those would be subjected to executive review? How many would be enticing for somebody to read? Would they bother? They'd say, of course something is wrong. They wouldn't have put it into the bug tracking system if something wasn't wrong.

So, there's no precision here. There's no invitation to read more. Let me illustrate the distinction. If my headline was: Cut didn't cut (200% zoom). That tells you more than it failed. It tells that you tried something and it didn't do it. And if I said: Cut the wrong area (200% and grow window). We have data corruption on the second one. Feature not connected might be a moderate severity bug, but for most folks data corruption is priority one. Crash or corrupt data about the same severity.

So we have a priority 1 bug here or least arguably a priority 1. And, does this look like priority 1? No, I don't think so. This is too vague to be a priority 1. This is just something is wrong with cut. But as soon as we say Cut cut the wrong area, Cut operated on the wrong data. We see something much more serious. In one case, they might not have hooked up the feature. In the other case, they hooked it to the wrong place. Bad.

So, the first piece that I want to suggest to all of you is that more precision invites, in cases where it's appropriate, more attention.

The second piece I want to mention is that most of you got the most important words first. Paint 95 is not the most important words. So probably that's not necessary. But if it was necessary, I'd probably still wouldn't put it in the front, I'd bury it in the back because I only a few words before this person looses attention. I say loses attention, you know when you look at this room, you don't see the whole room. You only see a little part of it. There's only so much visual angle that goes into your eyes. When you look at part of the page to look at the words, you see a few words, the rest are blurry. To get to the next words, you have to move your eyes. Your eyes move in jerks when you're reading. You focus on a few words, jerk, you focus on a few more, jerk, you focus on a few more. Every time you are going to the next few words, you are making a choice. It isn't smooth skimming. It's look, look, ah, don't bother.

And now we have the decision, is it worth reading any more? We don't know. I'm not trying to be critical, well I am trying to be critical in a friendly way, just trying to get where are the decision points where I entice the person to read further or don't.

Now from there, at least two function failures - at least you are identifying which the other folks didn't that there are two failures. But I would have written the two failures as separate reports instead of trying to make them here is something, you're not telling them what happened, you're telling them something is wrong with cut.

So, specificity is the first piece. The second piece in terms of conditions, cut doesn't work in all versions. The vice president in charge of being upset by bugs, reads this and comes out going; cut doesn't work, OH, NO. And now sitting outside this vice president's door is the programmer who's role is to calm down the vice president in charge of panicking so this person runs out of his room and says, how could cut not work this late in the season.

The programmer says calm down, calm down; let's take a look. Brings up the program, says, hmmmmm, let's draw something on the screen, them we'll mark it, then we'll cut it, see it cuts. Yup, no problem. And the vice president goes back and mutters because cut does work at least under some circumstances and if I read this I would think that cut never works. This is an example of a case where there's a critical missing condition.

Now you don't want to write all the replication conditions on the thing, but where you know that it's restricted narrowly, I would end up – my way of doing it is to put it into parentheses, *Cut doesn't cut (zoom 200%)* if in this particular organization they wouldn't understand that, I'd put: *Cut doesn't cut (interacts with zoom)*. That's okay and so the vice president runs out and goes, cut doesn't cut when I zoom, that still might be panicked, but now it doesn't get dismissed immediately. That's not the most important piece. I wouldn't put: At zoom 200% cut fails. This is the right order, get the impact first then your limitations your disclaimers. And I keep my disclaimers as short as I can (zoom).

Now I said I was going to have four sections. My four sections are:

1. how did it fail,
2. what were the interesting conditions,
3. what do I think was important (well there's zoom for the first one, and zoom and grow window for the second and maybe freehand selection),
4. then there are other conditions.

What are the other conditions? They're the ones I don't know are relevant but which are common to both and before I forget them, maybe I'll write them down. Where is this going BTW, this gets written into my nice, happy tester's notebook that no one is ever going to see except me or it gets written on my local stationary supplied in the cafeteria, right?

In my case, it gets written in my notebook. I just draw a little chart and I say, what did I see, what do I think some of the other relevant conditions are, what are some of the other conditions that were significant? Well, here are a few. *It was in the lower right hand corner. It was against a gray background. It was a circle. It was selected with the freehand tool.* It was today, Sunday. Could be a bunch of others. BTW, it turns out it doesn't matter if it was a circle but it does very much matter that it was the lower right hand corner. You'll get something very different if you put your circle somewhere else.

So other conditions are interesting. A lot of what you'll do when you are troubleshooting is to vary out the other conditions.

So you'll say, gee, does it matter if it's in the lower right hand corner? I don't know. Let me try it in the upper left hand corner and see what happens. And if you do that, you'll see that you get a squiggly little area that gets cut that is not the nice low __________ part, it's something that's no longer on the circle, it's off to the side and it's certainly not the full size of the circle, but it's a blotch. Move the circle over a little bit and now the squiggly little blank spot goes to the left of the circle instead of to the right. You move the circle up or down and sometimes the patch of blankness gets a little bigger, sometimes it gets a little smaller, sometimes it's up sometimes it's down but it's junk that happens somewhere. Who knows what this is? It's a floating deletion that you get to see as a white spot in the gray. Do you need to have the gray? No, you don't but you won't see white spots very clearly if you don't have it. But if you take the gray out and do the replication, then you'll see every now and again in this nice circle of black dots, oops a few of the dots disappear. So, it's still going away, it's just harder to see.

So, what are the other conditions and then notes. And then one of my critical notes is, it doesn't happen, cut cuts just fine if I move before I cut. And so in describing the first bug report, I will have my replication steps and at the end of the replication steps, I'll have a second column called notes and I'm going to say: note, if you do moving before cutting then it cuts normally. If you just do moving, it cuts normally. It's only cut before moving when you see the failure.

That's the kind of feedback I give to folks in class. In class what I find is that many people give much longer summaries. I get summaries that we have a lot of fun with them sometimes where I have a nice long whiteboard and I go all the away across the whiteboard and I go wait, wait, wait, and I go back to the start of the whiteboard and I keep going and if they speak to fast and it's too long, I get to go, wait I had a stack overflow, what was that again? And eventually they get the whole thing out and I say that's great. And I count out sixty characters and I cross out in red everything after that and I say, this is what they don't see. Now let's take a look at this report.

And I get to read something that is equivalent to: Paint 95: at least two different – that's the summary. Now yours is within sixty characters I'm just showing a proportional subtraction. Doing that kind of exercise gets folks to say, oh that's what's going to be seen. They are writing with the expectation of what's going to be seen.

We haven't been trained, as computer scientists; we haven't been trained in writing for other people to read. We haven't been trained to persuasive writing. In fact, I could ask the question, how many of you have taken a course in persuasive writing? Three, proportionally speaking, that's a little higher than the average in testing groups. But it's certainly not the majority in this room. In persuasive writing, you learn to ask the question, who am I trying to influence and how can I best say it for them? In technical writing, quite often what you're asking is how do I get the information across in a way that is accurate? Our writing is both, persuasive and technical. If the only thing that is technical, we make mistakes like saying too much and losing the reader.

So those are the things you can get to the headline with and the key things that come across to the students are:

- That they have to write for an audience.
- That the audience will pay attention to limited numbers of things so they have to pick the information they are going to give first.
- That they have to convey the information fairly and credibly or they will get ignored.
- And, of course, they have to not be vague.

We also go through a much longer exercise where everybody writes out the entire bug report.

(BTW, everybody will not write out the entire bug report because some people don't have English as their first language and to halt the class long enough for the slowest person to write the entire report would be disastrous. Give them an extra 20 minutes, some of them will write a full report and some of them will not. But show mercy at the end of about 20 minutes and stop it. And then take it up.)

There are a lot of ways to take it up. My most effective way is not to have people put their bug reports up on the board, because I'm going to talk about people completely misunderstanding the problem and that would be really embarrassing. Here is just bad wording. I haven't said anyone is dumb or missed the point. If I'm going to do that, I want it to be very anonymous.

What I've done over the 20 minutes is to look over everybody's shoulder and ask, what are you writing? Here's the most common single mistake: When I was describing the bug, every time there was a new test I said now get out of the program and come back in, color the background gray, draw the circle, you know I went through that nice little sing song. You will be surprised how many students will forget that. And when they write how to reproduce a bug they will start with the very first test on slide 2 – color the background, freehand select, then draw a circle, freehand select again, cut the circle, color the background again, draw the circle again, and on they go until they finally say, and now the circle doesn't work.

Are all those steps relevant? No, we exited the program, we came back in, we got a new screen. It shouldn't make any difference. Why are they doing this? Because in the written stuff it never says exit the program, come back in. It was only auditory. We process information in two channels, we process a lot of channels, sound, smell, sight – for most people, sight is dominate. If there is visual information and there is auditory information, visual will win over sound. What you see is a description of the report in a certain sequence. What you heard was a modification of the sequence.

But especially if you give people over lunch or over night, a few of them will forget the sequence and will just write the entire thing. The longer you wait, the more people will not do it. If it is a short break or a short lunch, may be only 1 person will not do it. You don't want to subject that person to ridicule, it's very important that you wander around the room and look and see the patterns of mistakes yourself. And then say, here's an example of something somebody did. It's also important to not say, here's an example of something some fool did. Because they'll know you got it off their paper and they won't appreciate it.

Here's an example of something somebody did, they started on the first slide and they went through the entire sequence. Now why is it relevant to set that thing up in the first place? It's relevant because situations like this happen in bug reporting all the time. Here's the scenario that testers run into regularly. You write a bug, it goes to the programmer, the programmer calls you up and goes, I don't understand. You walk over to the programmer's cube, you show the programmer what happens or you tell the programmer what happens. The programmer goes, oh, I get it now. And then you go away. And tomorrow the programmer starts working with the bug again. And what she sees is the written description of the bug. And what she doesn't remember is exactly what you told her in the supplement.

So with that bug, it's unlikely the programmer will call you and say I just don't remember what you told me yesterday, would you mind coming back to my cube and telling me again. More than likely she's going to react to this bug in an embarrassed way and the embarrassed way she's going to react to it unless it's a fatal bug is say, she's going to react to it by saying it's a feature, it's not a bug, I cannot reproduce, no one would do this; which all really means, rats, I'm too embarrassed to call up and find out what it was, I'm just going to make it go away.

When you get someone calling you and telling you there's missing information, they're giving you a bug report against your bug. The fix is not to walk over and fix it with the person, you have to do that too to make interpersonal nice/nice, but the real fix for the report is to go into the report and fix it. Add the extra step or add the qualification or add the explanation, if it's not in writing, if it's not in the same place, the same medium as the dominate information, it will be forgotten and it will not be linked when somebody reads it again it just won't tie together.

So just like some students will forget all of those obvious cues, a reader who is told and demonstrated what the problem is will forget that or will have a reasonable non-zero probability of forgetting the detail if it's not in the report itself.

So if you write an incomplete report, the fix is not to explain what else needs to be done person-to-person, the fix is to explain in the report itself exactly what needs to be done. And then to walk over to the person and say, yeah, I updated the report let me show you what I showed you the first time.

I don't know if my visual was overriding my auditory, but I could swear I heard you say, "write your headline" not "headlines" and that felt to me like a constraint to come up with one and only one which then I felt like, don't criticize me on not writing two, you told me – if I was the angry student I would pop off at that point. And the second thing I'd do is say all right, would you please critique the last two in the student notes on the same basis that you critiqued ours.

So when I tell people to strike a bug, I know exactly what you're saying. In most classes I get a substantial number of people writing two bugs anyway because I throw them two failures with two bugs. You guys follow directions too well. I have actually never had anybody come back to me and say, but you told me to write only one. They might have felt it but you are the first person who raised it as an issue. I'm not saying that it's a bad thing.

I've had a lot of feisty students. But the sharp, feisty ones express it by giving me two reports. And often how that works in a class where there's a little more time, I wander around the room, I ask question.

I typically do this in a 20 minute session where I say take 20 minutes and out of that make sure you reserve 10 minutes for yourself for a break and we'll come back together at the end of 20 minutes. That way, the people who can't write anything in 10 minutes, can take 15. Now in that period I wander around and people are likely to ask questions like, do I really have to stick with one? And I go, do what's right. That might be the distinction. Almost every exercise I do in the professional classes that I teach do coincide with a break so that I have an extra 10 minutes for people who's first language isn't English so they can write it without everybody else staying. We still have to wait for slow writers, but not as much.