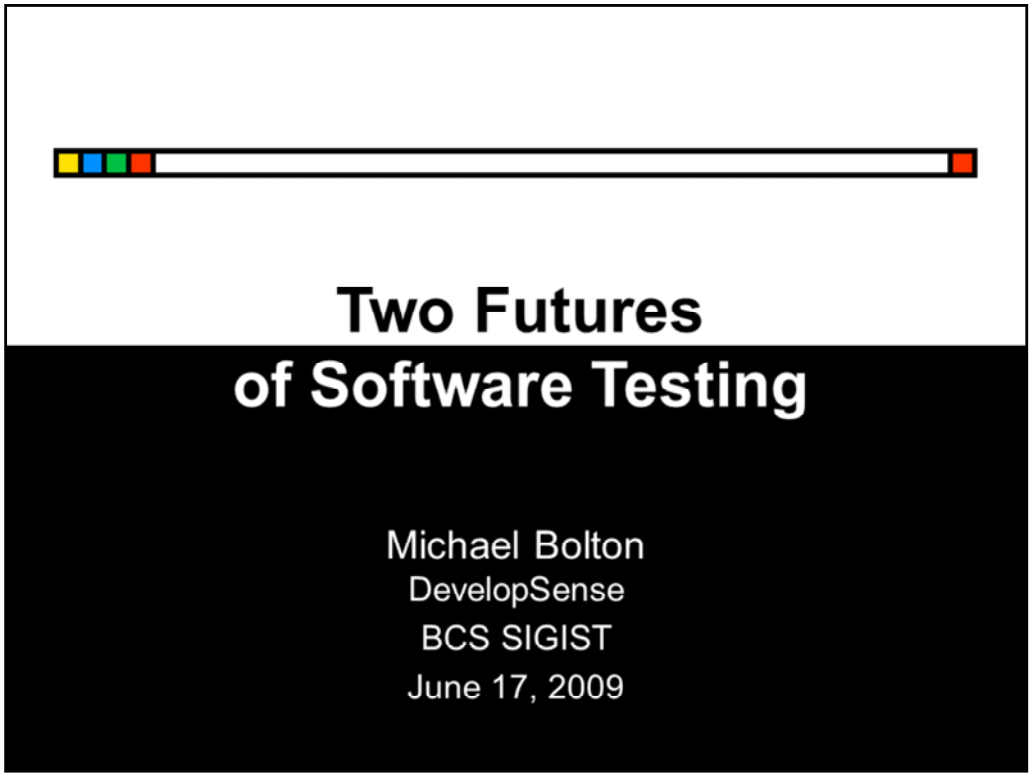
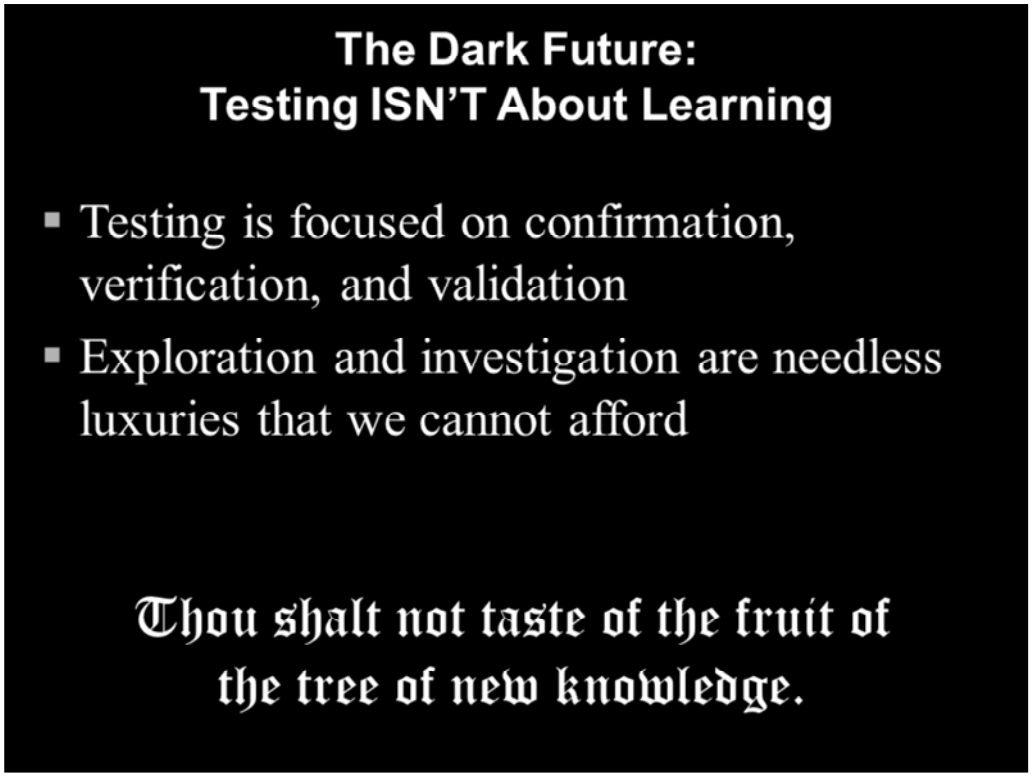
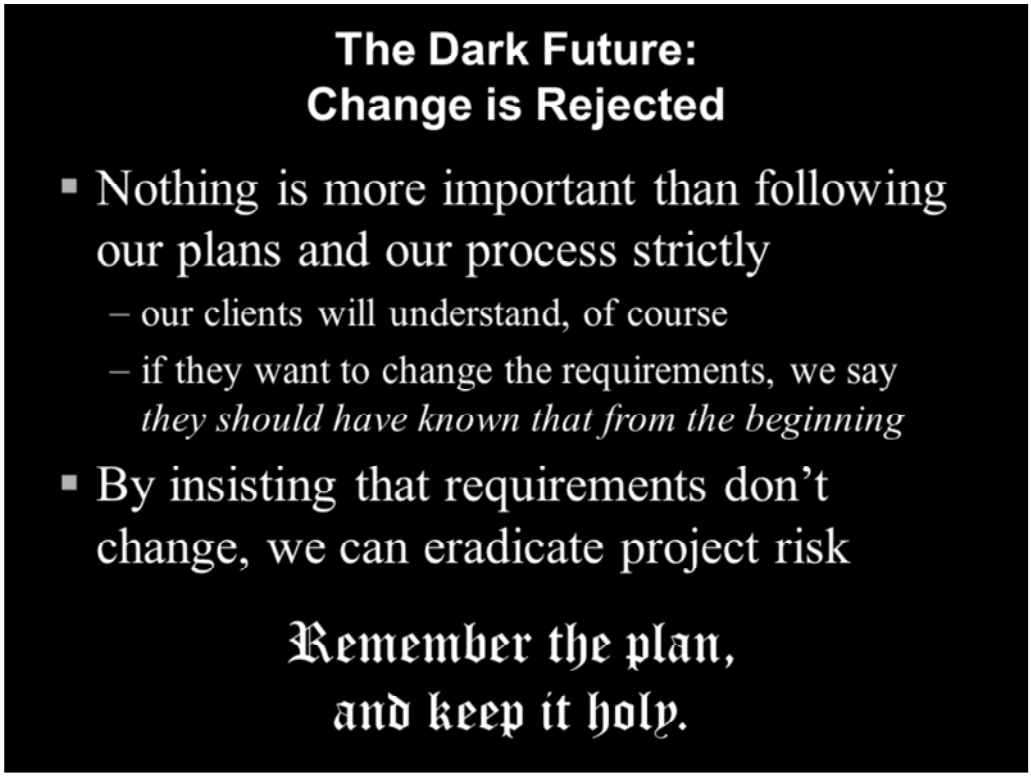
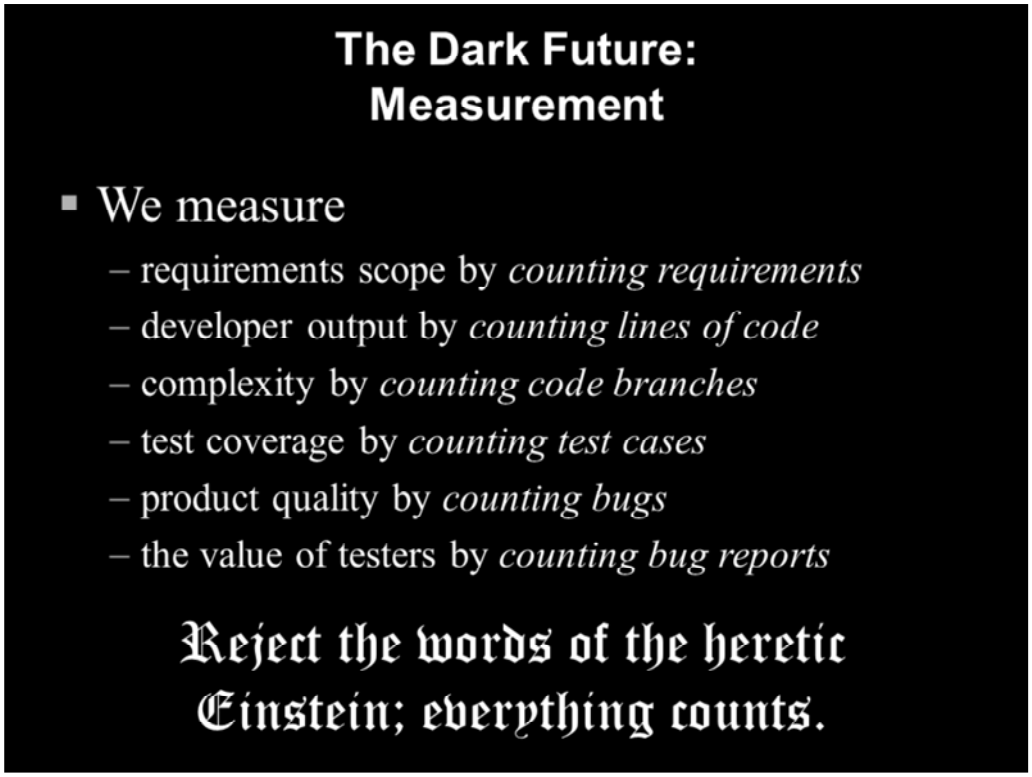




In the Dark Future, testing is a relentlessly routine, mechanical activity, even when it's done by humans. It's not about learning, it's about confirming things that we already know, answering questions for which we already have the answer, repeating the same mindless tests over and over again. There's no place in the Dark Future for exploration, investigation, or discovery, or learning, and that means that there's no place for skill, creativity, or imagination. Nor is there room for asking questions about the customers and how they might value our product. We just do what we're told, and we learn nothing.



In the Dark Future, it is the role of the tester—excuse me, the *Quality Assurance Analyst*—to inhibit change. Change brings a chance of invalidating things that we believe we know about the product and the project, and thus change involves risk. So even though the customer needs, the market conditions, the schedule, the budget, the product scope, the staff, and everything else about the project might change, we should stay the course and stick to the plan. It doesn't matter if we learn things through the course of developing the product; we should have known those things beforehand. It's not merely our job to inform; we must also *enforce*.



Requirements, productivity, complexity, test coverage, product quality, and tester value are influenced by dozens, hundreds of factors that we could observe. Yet most of these factors are not tangible or countable in a meaningful way, and simplistic attempts to count instances of them are practically guaranteed to mislead. In the Dark Future, we make these problems go away by *ignoring them*.

A requirement is not a line or a paragraph in a document; those things are representations—literally *re-presentations*—of the difference between what someone has and what someone desires. Counting a requirement by counting a line in a requirements document ignores everything about the meaning and the significance of the requirement, like counting tricycles and space shuttles as equivalent. Despite this, we'll simply apply the idea that there should be one test case traceable to each requirement. No, wait! Two! One *positive* test case and one *negative* test case.

A line of code is a representation of an idea. A line of code can be as simple as placing a value in a CPU register or as complex as a multi-branch, multi-condition decision point. A developer's job is about learning, solving problems, and shaping and reshaping solutions. Sometimes that means removing lines of code rather than adding them. There's far more to a developer's job than counting the number of characters that she's typed. Lines of code are just scaled-up versions of the same silly measure.

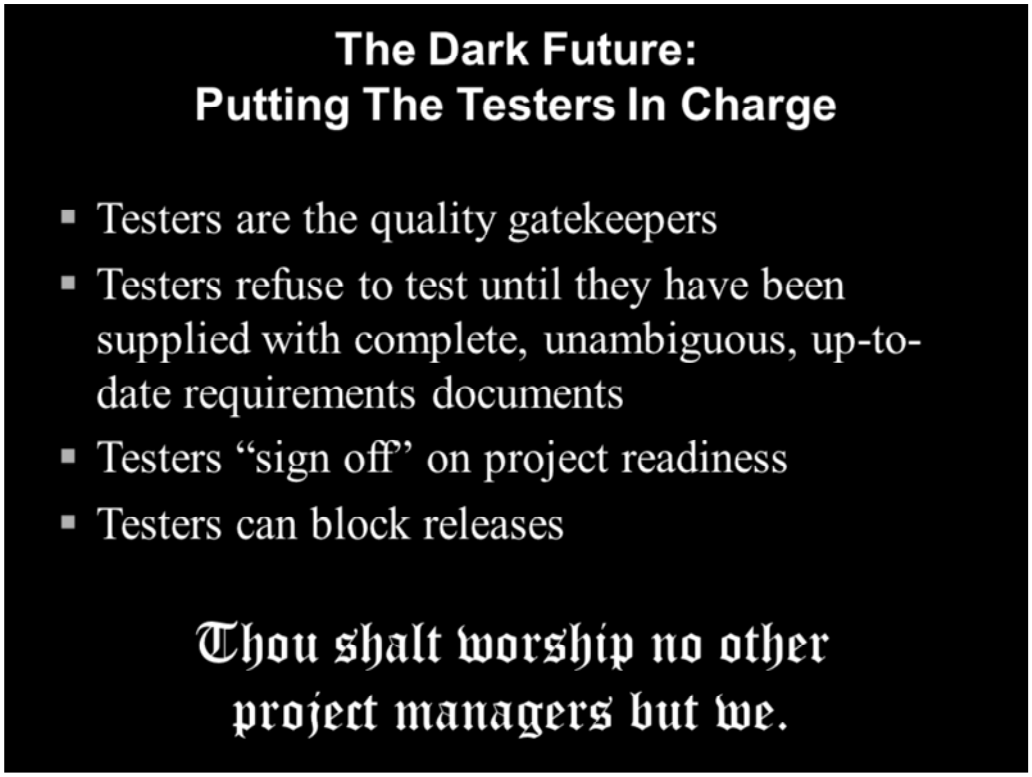
Cem Kaner has said that a test case is a question that we want to ask about the product. As James Bach has said many times, a test case is a container for a question. In the Dark Future, we'll evaluate the quality of work in an office by counting the briefcases that come in the door every morning. We won't bother to look inside them. If more briefcases come in, it's obvious that the quality of the company's work will improve.

A bug is not a thing in the world. A bug is a *construct*; thought-stuff; a mental thing. It's a relationship between some person and some product, such that some other person might not view it as a bug. Even when two people or more agree that some behaviour seems to be a bug, they may disagree on the significance of the bug. Despite this, in the Dark Future, we'll just count 'em. More bugs means higher quality; fewer bugs means lower quality. That applies to testers too. We'll ignore all the other activities and dimensions of value that a tester might bring to a project, and count their bug reports to measure their effectiveness.

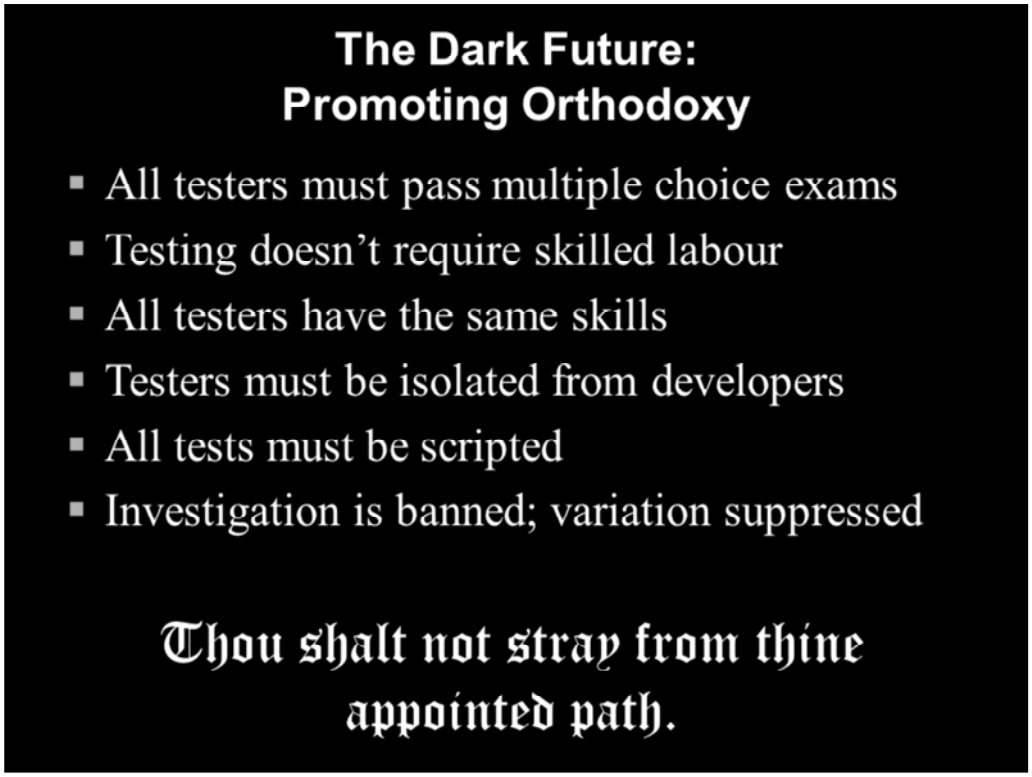
We'll certainly ignore problems associated with simple metrics by avoiding *Software Engineering Metrics:*

What Do They Measure and How Do We Know? by Cem Kaner and Pat Bond (http://www.kaner.com/pdfs/metrics2004.pdf); the classic *How To Lie With Statistics*, by Darrell Huff; *Quality Software Management, Vol. 2: First Order Metrics* by Gerald M. Weinberg; and especially *Measuring and Managing Performance in Organizations*, by Robert D. Austin.

Einstein said that "not everything that counts can be counted; and not everything that can be counted counts." We'll ignore that too.



In the Dark Future, testers are in the driver's seat. It is we who decide whether the product should ship or not. It is our signature that managers must obtain to be sure that they're shipping a quality project, and our permission that they must obtain to release it. We decide when to start testing, and we do so only when the product and the accompanying documentation adhere to our rigourous standards. We're not obliged to follow these standards ourselves, of course; that's not our role. In the Dark Future, our role is to tell other people what they're doing wrong and how to do it right.



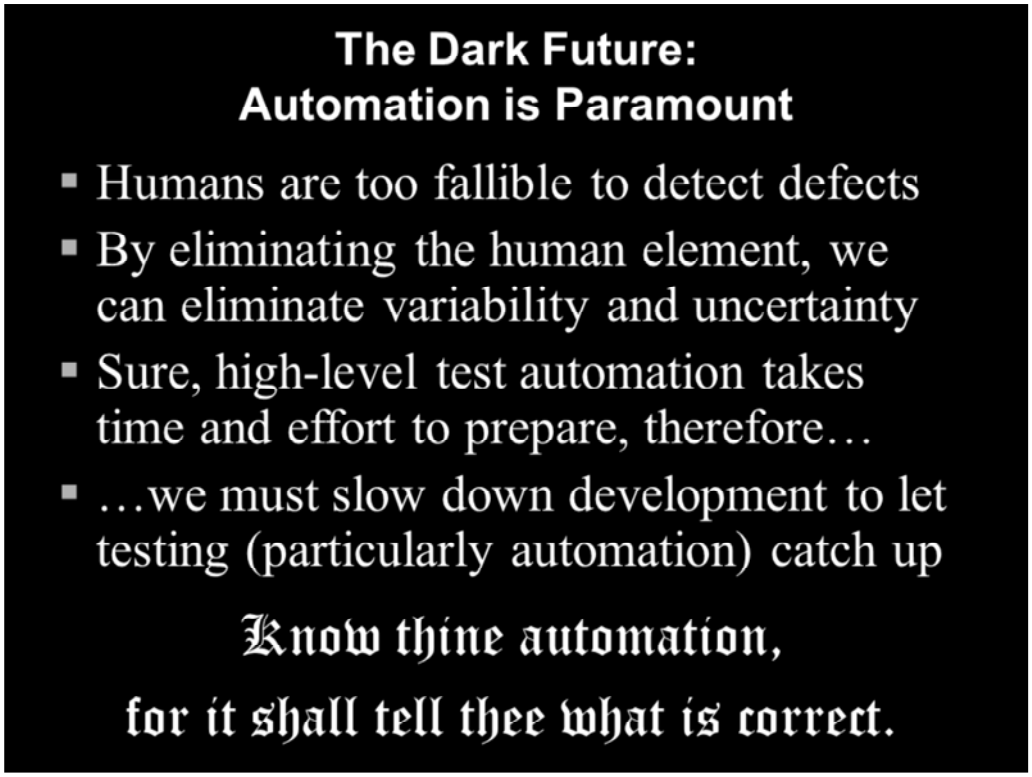
In the Dark Future, testers will be evaluated based on their ability to memorize testing terms from a particular authority's body of knowledge. Context or interpretation have no place in the Dark Future. Exams should always be set up for the convenience of the certifiers, so multiple choice is definitely the way to go here. If there are concerns that this approach is insufficient to evaluate skills, no worries: testing isn't an especially skilled trade anyway. There will be some testers who are able to write code, but testing is mostly an uninteresting, repetitious, confirmatory task anyway.

We don't want testers to be hobnobbing with the developers (that is, the programmers—but programmers are the only developers in the Dark Future). Testers are too weak-willed to avoid the pernicious influence of programmers, so mingling might compromise the testers' objectivity. Testers might even be tempted not to report bugs.

Repeatability is very important in the Dark Future. We want to run the same tests over and over, without variation, because variation might lead to unpredictability. Discovering and investigating bugs could throw our whole schedule off.

In the Dark Future, ISO Standard 29119 will tell us what to test and how to test it. *"Whatever type of testing you do, it will affect you."* It doesn't matter if the people who drafted the standard know *your* business; they're experts, and they know better than you what's good for you. *"The standard uses a four layer process model starting with two organizational layers covering test policy and organizational test strategy. The next layer moves into project management, while the bottom layer defines the fundamental test process used for all levels of testing, such as unit testing, system testing, acceptance testing, and the test types (e.g. performance and usability testing). Parts 2 and 3, on process and documentation respectively, are particularly closely linked as all outputs from the test processes potentially correspond to documents defined in the document part. There is also a 'new work item' being suggested that would see a fifth part initiated on test process improvement – imagine a testing industry without the emergence of another new test improvement model every couple of years."* Doesn't that sound swell? Not only will they be telling you what to do, but also how to improve it– despite the acknowledged caveat, *"Probably the biggest complaint raised against IT standards is that they do not meet the needs of actual practitioners – most of us have come across such 'shelfware'."* Don't worry about the standard being unmanageable, either. The current draft of Part 2 of the standard is, as of this writing, a mere 100 pages.

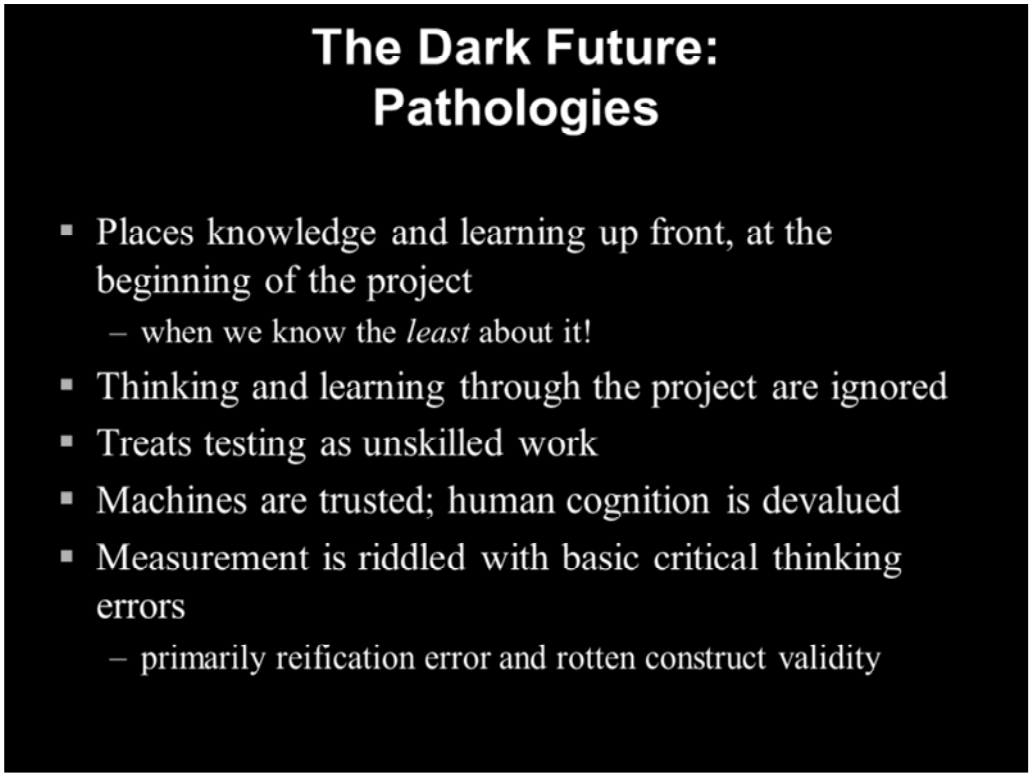
Note also that there is a standard vocabulary associated with the standard. That standard vocabulary will be in English. Translating it into other languages will only increase complexity and ambiguity. Let's all just test in English. If other cultures don't like that… well, tough. There's not much to learn from them anyway.



The Dark Future puts automation at the centre of the testing process. After all, computer software runs on computers, so what better than a computer to make sure that everything is okay?

In the Dark Future, we will again ignore problems with our assumptions. First, we'll ignore that there are vastly more factors in the success or failure of a computer program than functional correctness. We'll also ignore the fact that test automation is itself a software development project, subject to the same kinds of problems as the applications we're testing. We'll ignore any potential for confirmation bias—that we'll tend to run tests that confirm our beliefs. And we'll ignore automation bias—the bias associated with being convinced that something is so because a machine says it's so, and to be blinded to problems that automation doesn't tell us about.

Most significantly, we'll focus on correctness and not on value.



Many dark visions of the future devalue the skill, freedom, and responsibility of the individual tester in various ways.

Dark visions devalue the significance of *learning*, which is of the essence of software development, and which is one of the primary tasks of the tester. A principle of the context-driven school of software testing is that projects unfold over time in ways that are often unpredictable. The planned product and the actual product tend to diverge from one another, starting on the first day of the project, and getting farther apart as we learn new things. One option is to prepare a detailed, heavyweight test plan, and then to spend a large amount of time updating it—time that might be better spent on interacting with and investigating the product. Another option to do a lot of planning in advance, and then abandon it and fly by the seat of our pants. A third option, though, is to recognize that we are going to learn something new every day, and so to produce lightweight, adaptable plans that can be updated quickly and easily. As Cem Kaner has pointed out, why do a huge amount of wasteful preparatory work at the beginning of the project—the very time that we'll know less than we'll ever know about it?

Reification error is the practice of taking constructs—thought-stuff—and mistaking them for tangible things in the world.

Construct validity is the degree to which our metrics and measurements accurately reflect real attributes of the thing that we're measuring. When so much related to software development and quality is based on subjective perception, construct validity is very difficult to achieve when we use numbers alone. That's okay; in the Dark Future, we'll ignore that problem.

In the Dark Future, testers have blame without responsibility, culpability without authority. Since they were the last people to have their hands on the code, it is assumed that any undetected problems are their fault. Testers are required to sign documents asserting that the product is acceptable for release, even though the release of the product is a business decision, rather than a technical one.

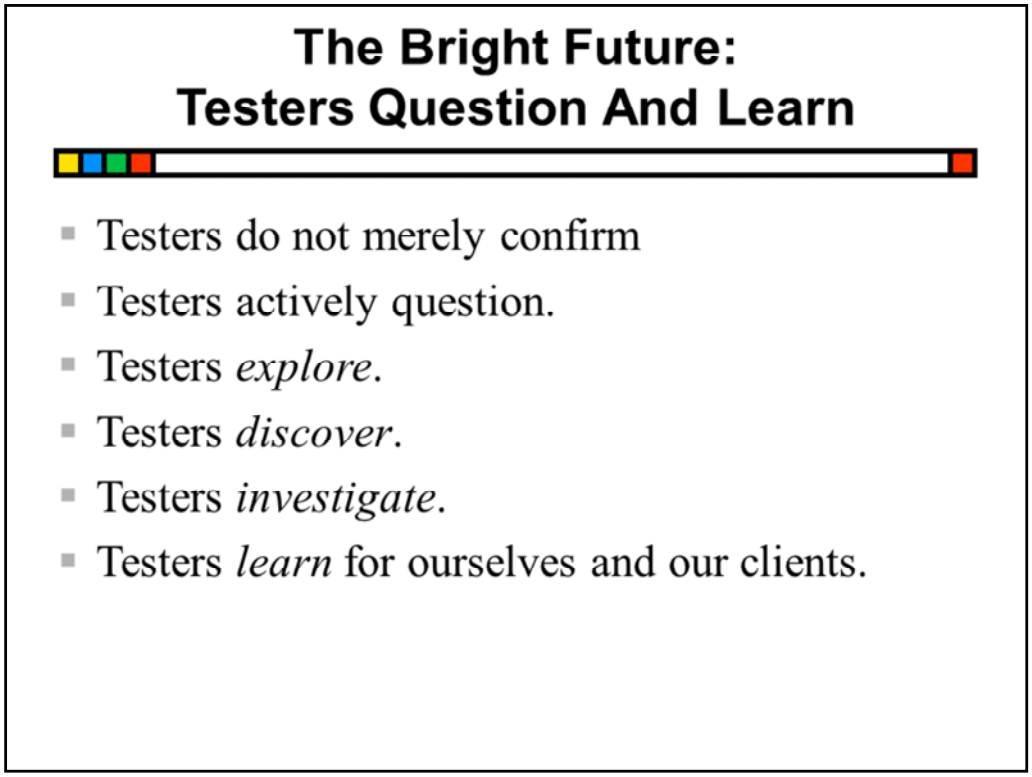
In the Dark Future, all product failures are seen as testing failures. There's no recognition that problems are problems for the whole development team. Read the paper, and you'll see over and over that problems are pinned on poorly tested software. Not poorly programmed software, nor poorly managed projects, not poorly conceived products, not poorly developed requirements. Product problems are testing problems; no more, no less.

The Bright Future
Testers Light The Way

This is our role.

We see things for what they are.
We make informed decisions about quality possible,
because we think critically about software.

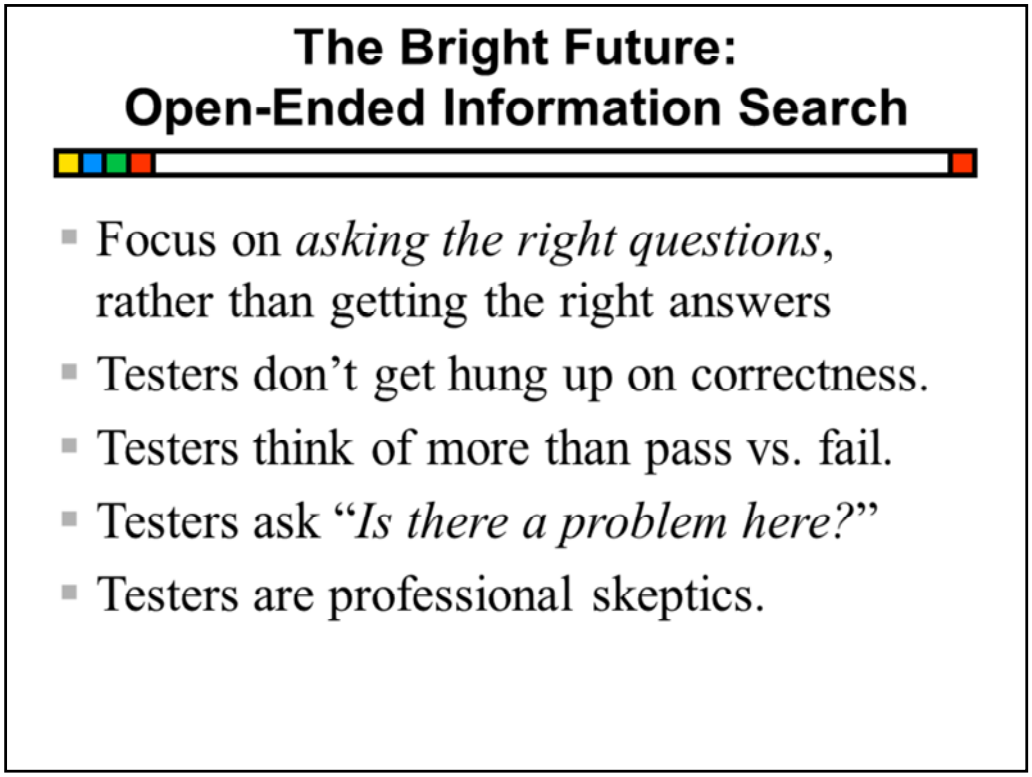
We let management make those decisions.



In our Rapid Software Testing course, senior author James Bach and I define testing as *questioning a product in order to evaluate it.* Cem Kaner defines testing as "an empirical technical investigation of the product, done on behalf of stakeholders, with the intention of revealing quality-related information of the kind that they seek. These definitions are entirely compatible. One is more explicit; the other is shorter.

Confirmation is about answering the questions "Does it do what we expect?" or "Is it *still* working?" In general, such tests tend towards the scripted end of the exploratory/scripted continuum. In general, we ask and answer the question "*Can* it work?" with tests that tend to be more exploratory and more challenging. Still more challenging and more exploratory tests probe the product, asking "*Will* it work?" and "Is there a problem here?" Part of our job is to help defend against anticipated problems—but another part is to discover problems that no one else anticipated.

We focus on risk (but we do some testing that isn't risk-based, the better to discover new risks). We continuously develop new questions (but we're prepared to stop testing at any time that the client withdraws the mission). We take testing to be primarily investigative (but through investigative tests, we recognize that we're also getting confirmation). We're delighted when programmers handle the bulk of the confirmatory testing effort at the unit level, where automation is inexpensive and feedback is immediate (but if programmers aren't doing that kind of testing, we inform management of the risk and test accordingly). We develop skill in exploratory testing, using concise documentation and rapid feedback to our clients (but that's not to say that we reject ideas about oracles and coverage from other sources). We ensure diligently that our work is entirely accountable and stands up to scrutiny.

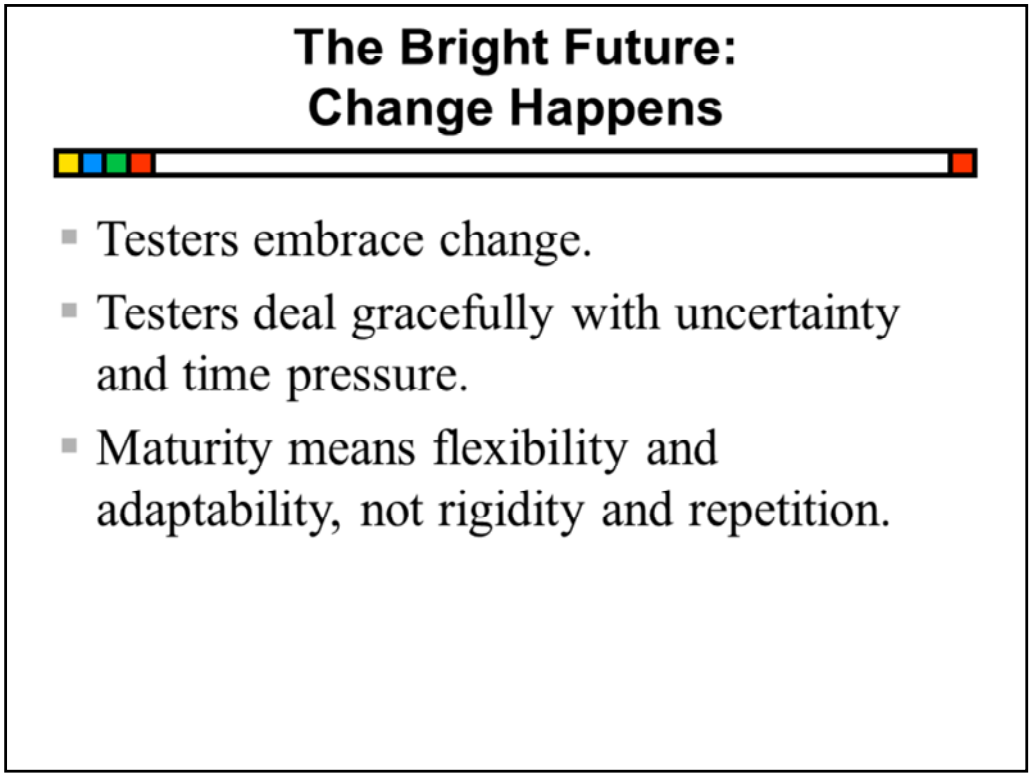


When we're doing excellent testing, we're learning things about the product on behalf of our clients and the project community. This means much more than knowledge about functional *correctness*. To a great degree, in the Bright Future, the programmers will take care of a lot of that, through improved design and unit testing. In the Bright Futures, testers will be guided by Cem Kaner's definition of a computer program (http://www.kaner.com/pdfs/KanerSocialScienceSTEP.pdf). Kaner says that a computer program is not merely "a set of instructions for a computer." Instead, a computer program is "a communication among several humans and computers, distributed over distance and time, that contains instructions that can be executed by a computer." In the Bright Future, testers will evaluate the quality of that communication and its value to stakeholders. We'll recognize and test for the many dimensions of quality—capability, reliability, usability, security, scalability, installability, performance, installability, compatibility, supportability, testability, maintainability, portability, and localizability. We'll investigate and probe the product and its relationships to the systems and people with which it connects.

We'll remember that the product is a solution to a problem, and if the problem isn't solved, then the product doesn't work. Confirmation, verification, and validation are important, but they're mostly about *checking*. Checking is especially important, especially useful, and especially inexpensive when programmers are doing it as part of the process of writing a program. *Testing* in the Bright Future is something more that just checking. Among other things, testing means actively engaging with the product, interacting with it, providing it with challenging inputs, seeking extents and limitations, exercising it with complex scenarios, giving it a lengthy set of tasks—one thing after another after another. As Jerry Weinberg points out in his book *Perfect Software and Other Illusions About Testing*, testing is also about challenging the constraints of the situation to obtain maximum relevant information for our clients at maximum speed. Often it's about taking notice of things that are hidden in plain sight; listening to the way people answer questions in addition to the content of the answer; observing the way people interact with one another; and watching for the ways in which they might be gaming the management systems.

In the Bright Future, testers are skeptics, not cynics. In his forthcoming book, *Secrets of a Buccaneer Scholar*, James Bach points out that skepticism is not the rejection of belief; it's the rejection of *certainty*. We're hired to help our clients identify risks and become more confident about their understanding of the system under

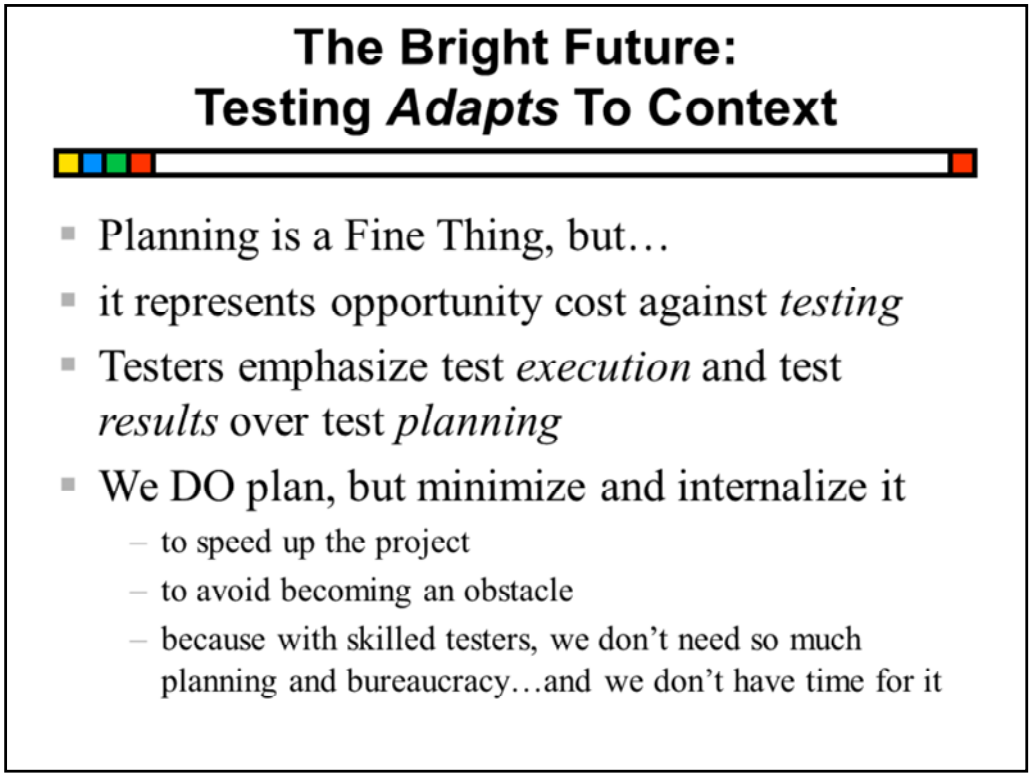
test, but we can't promise absolute certainty. Part of our role is to maintain productive doubt when everyone else is certain that there are no problems here.



Change happens. We better serve our clients when we recognize that changes are their prerogative. We recognize that no one can know everything there is to know about the product in advance of its being written. We recognize that market conditions change, and that our projects and our products must adapt. We recognize that, as we develop and test the program, we will discover requirements of which the development team was previously unaware. We try to help the project community to anticipate what might be on the horizon, but we understand that things happen.

We're amenable to management's decisions about when to ship the product, whenever that may come. Our role is to inform management of what has been tested and what hasn't been tested; of what we know and what we don't yet know; of what further questions remain to be asked and answered. We can help to identify risks associated with missing or partial information, but the decision to ship remains with the business. Instead of trying to run the project, we focus our attention on investigating the product as quickly and as deeply as we can with the time and the resources we have available.

In a biological system, a mature organism is one that can fend for itself without resorting to parental protection. In software development, it seems that we have a different meaning for "maturity": doing the same things consistently. Shouldn't maturity be about adaptability and responding appropriately, rather than jumping up and down and protesting when someone doesn't do things the way we'd like?



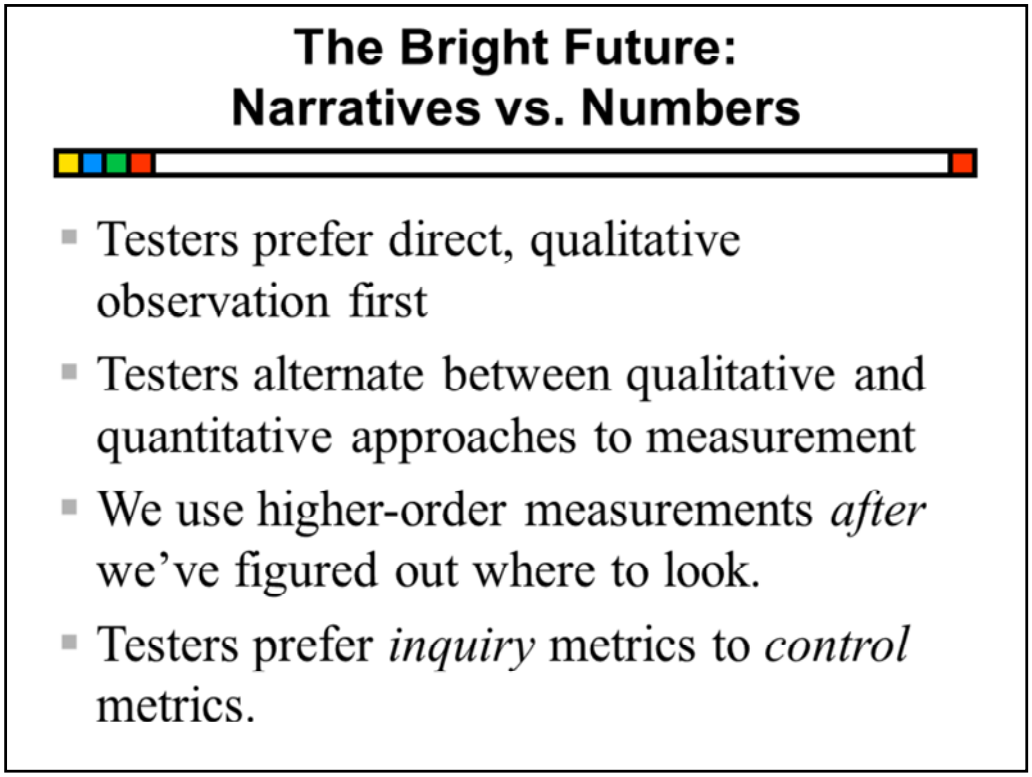
In the Bright Future, we'll have learned lessons from Karl Weick and his book *Sensemaking in Organizations*. We'll have learned that maps and plans animate and orient people; they're tools that help get us up on our feet, looking around, and trying to make sense of the world around us. Maps and plans might help us get started, but it's what people *think* and what people *do* that makes the difference. We'll learn from the past that it is thinking and doing, not the plan, that generates results and knowledge, and we'll learn to give credit to the people who learn, and not to the plan.

This is not to say that we do *no* planning, but rather that we do as little planning as is necessary to guide and accomplish the mission. Our focus is on interacting with the product, rather than detailing how we plan to do it; observing everything that might be relevant, not just an explicit expected result; using the result of the most recent test to help us decide what the next test should be.

In order to become good at this, we must become expert at chartering our testing concisely, recording our actions appropriately, and reporting important observations cogently. This means developing skills in note-taking, in the use of record-keeping tools, in dynamically managing the focus of our work, in rapid evaluation and estimation.

If the mission *requires* lots of documentation and data, we supply it, but we regularly check to make sure that it's adding value. I can practically guarantee that someone will leave this presentation claiming that I advocate no documentation, ever. I *don't* advocate that, and the document you're reading now is the **documented** proof.

I advocate no documentation *that wastes time and effort.* If the mission *requires* lots of documentation, we produce it—but we also question the mission, to make sure that our client recognizes that more documentation usually means less testing, and to make sure that we're clear on where the priority is. If the mission *requires* lots of automated testing, we develop it, but we don't stop brain-engaged interactive human tests. If the mission *requires* us to suspend our skills, we do, but we do son on the understanding that someone else is responsible for the quality of our work.



How do we measure the quality of testing or the quality of a product?

First of all, recognize the difference between *inquiry metrics* and *control metrics*, as James Bach and I often discuss in our Rapid Software Testing class. Inquiry metrics *prompt questions*. They point us to things that we could observe. Control metrics *drive decisions*. They assume that the metric tells us everything that we need to know, and direct us to change our path. Control metrics are dangerous.

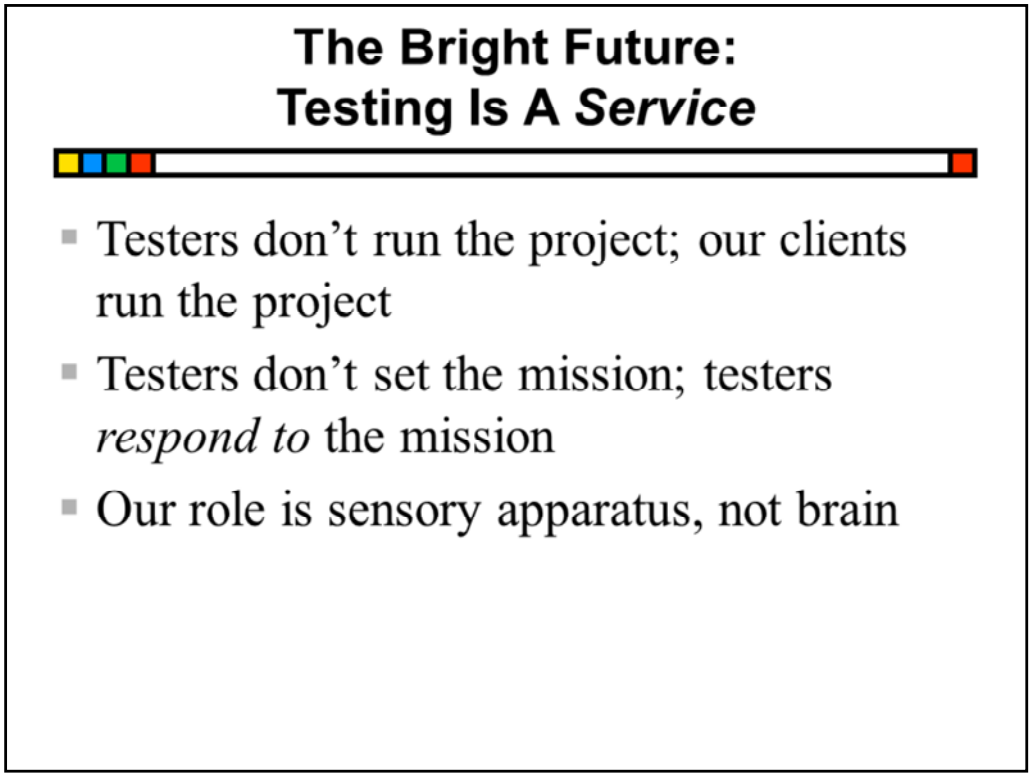
Second, prefer direct observation to more abstract measurement. Most kinds of software measurement are aggregations of data that end up obscuring information.

Third, prefer qualitative rubrics or balanced scorecards to unidimensional counts of things that are themselves abstractions. Read the publications above for more detail.

Skilled testing is storytelling. Journalism and investigative reporting are other disciplines from which testing could learn; Jon Bach was trained as a journalist.

Testers compose, edit, and narrate cogent stories *about the product*, about how it can work and how it might fail. Testers also describe how they got those stories, by telling stories *about their testing*—what they did, and why they did it, what they didn't do, and why they didn't do it, and about why the testing was good enough. Testers use stories to add depth and richness to operational models, use cases, and scenario tests.

Skilled testers and test managers encourage management to reject deceptive quantitative measures (for example, counting test cases, counting requirements, or counting bugs) Some people suggest that "the numbers speak for themselves." They don't, and if someone doesn't speak for them, the listener will spontaneously—and often unconsciously—invent a story to go with them. That's why skilled testers don't supply numbers without a story—and why we tend to prefer leading with the story, using the numbers to back it up.



Testers provide information to management so that management can make informed decisions about the product and the project. Testers don't *make* those decisions. We're not the brains of the project; we're the antennae. We don't drive the bus; we observe the traffic around us and the road ahead, and report to the bus driver. We're not the skipper or the helmsman; we're in the crow's nest.

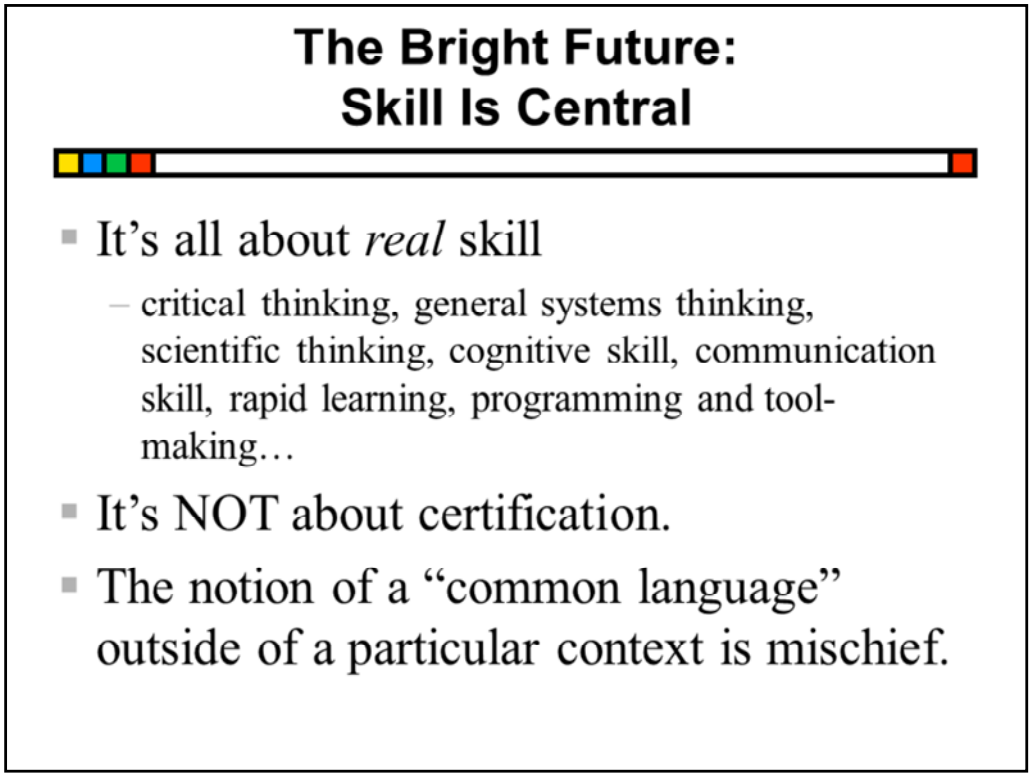
Testers don't write the code; we don't debug the code; we don't make changes to the code; we don't have control over the schedule; we don't have control over the budget; we don't have control over who works on the project; we don't have control over the scope of the product; we don't have control over contractual obligations; we don't have control over bonuses for shipping on time; we don't have control over whether a problem gets fixed; we don't have control over customer relationships. Other people, particularly programmers and managers, handle that stuff. With so little that's in our control, how can we be responsible for quality? Management, not testing, makes the decisions; management, not testing, has the responsibility.

Testing is not the control mechanism; testing is one of the feedback mechanisms. We *are* responsible for the quality of the information that we provide to management; that is, we are definitely responsible for the quality of our own work. But the idea that we're responsible for the quality of the product, or that we're the only voice of the customer, is ridiculous—and as Cem Kaner has pointed out, it's offensive to other members of the project community who can legitimately claim to be voices for the customer.

If you want superb discussions about quality and how testers interact with it, read *Perfect Software and Other Illusions About Testing*; *Quality Software Management Vol. 1: Systems Thinking*; and *Exploring Requirements: Quality Before Design*, by Jerry Weinberg. Read *Lessons Learned in Software Testing* by Cem Kaner, James Bach, and Bret Pettichord. Read Cem Kaner's brief article "I speak for the user: The problem of agency in software development," (http://www.kaner.com/pdfs/I%20Speak%20for%20the%20User.pdf) and his slide presentation "The Ongoing Revolution in Software Testing" (http://www.kaner.com/pdfs/testingRevolution2007.pdf).

Excellent testing is mission-focused. Cem Kaner and James Bach have enumerated a large number of possible testing missions. The testing mission might be focused on finding the most important problems in the product;

it might be focused on finding as many problems as quickly as possible. It might include assurance that the product is compliant with particular laws or specific standards. It might be focused on finding workarounds to known problems. The mission might be to assist with ship/no-ship decisions, to assist the programmers in developing test frameworks. Testing responds to these missions, whatever they might be.



The state of tester certification as of this writing should be seen as an embarrassment to the craft. Tester certification (in particularly that provided by the ISTQB Foundation Level Certification) is itself the epitome of *bad testing*. Its purpose is not to certify, but as Tom DeMarco suggests, it's to *decertify*—to shut out people who haven't shelled out money to the certification board for the exam (and optionally to its accredited training providers).

Tester skill is at the centre of all excellent testing. Important skills include

•critical thinking – recognizing bias and thinking errors;

•general systems thinking – coping with complexity, models, observation, and interactions between systems;

•context-driven thinking – coping with changing situations and values;

•scientific thinking – designing and performing experiments, and recognizing that knowledge of something is never final;

•cognitive skills – learning and using lots of observational modes;

•communication skills – writing, recording, and reporting;

•rapid learning – leveraging all of the means at our disposal, including exploration, to increase understanding; and

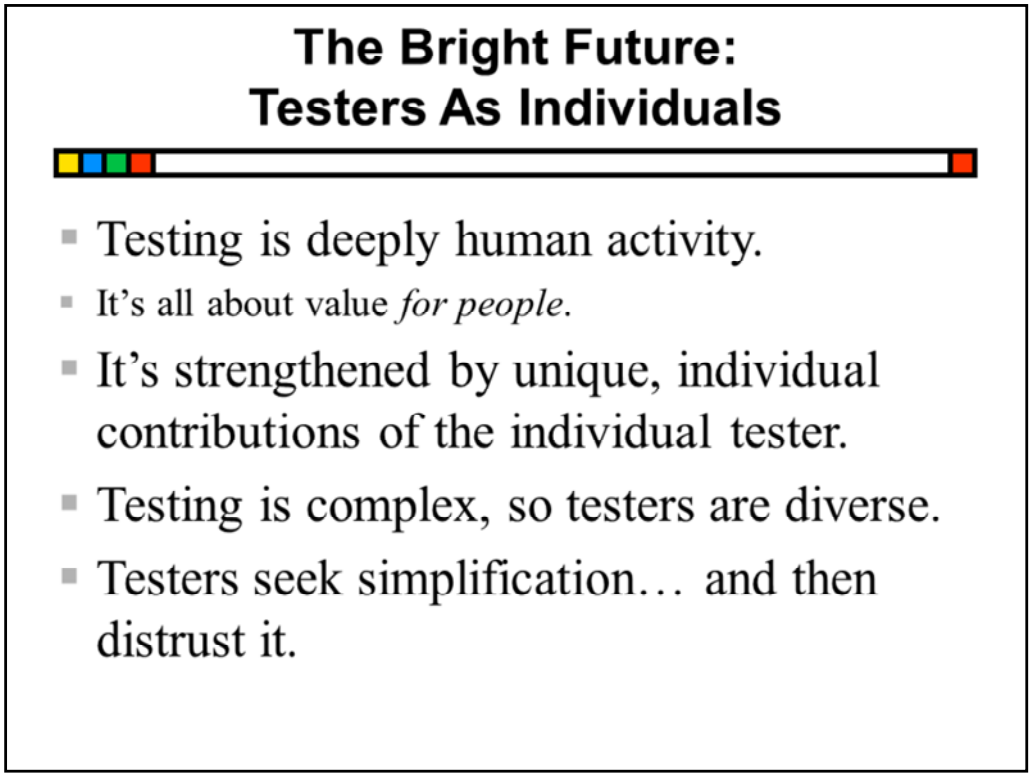
•programming – putting the machine to work for us so that we can create our own tools.

Obviously, this is not even close to being a comprehensive list. Obviously, different testers will have different interests, different skills, and different skill levels. That's all to the good. Diversity of skill brings diversity of approaches, and that's helpful in finding many and varied bugs in many and varied products.

Over the years, there have been calls—mostly from certifiers and process enthusiasts—for a "common language" for testing, as part of increasing the "maturity" of the profession and making testing a "disciplined profession". Testing glossaries are prescriptive, attempting to define abstract concepts with one (and usually

only one) definition. By suggesting that there's one, and only one right answer to testing questions, the exams run counter to Jerry Weinberg's observation that a tester is *someone who knows that things could be different*.

Testing always takes place within a context. Consider terms like "integer", "rounding", or even "testing". These terms mean different things to different people, depending upon the context in which they're used.



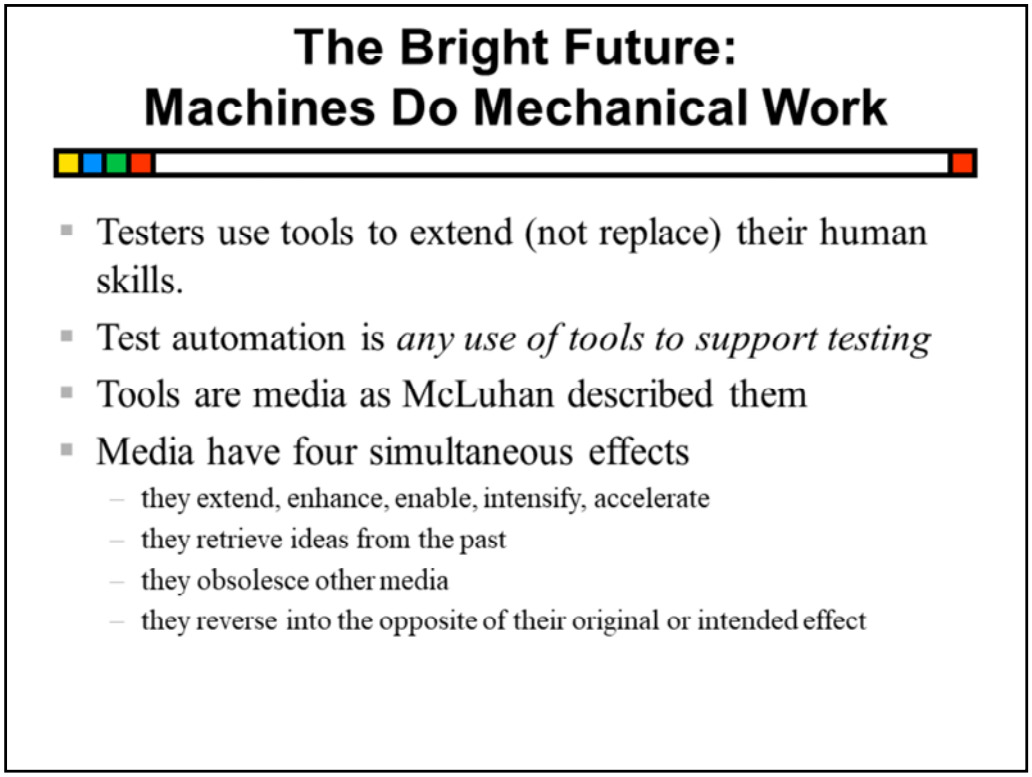
In the Bright Future, we emphasize *humanity*. We focus on the fact that our products are being developed to help real people solve real problems. We recognize that each tester brings a unique perspective, a distinct set of skills, and a particular suite of experiences to the table.

In the Bright Future, helping the managers and the project community to understand value is one important role of the tester. (We don't say "the most important role", because we're aware that changing contexts mean changing priorities.) From one perspective, testers don't add value; we don't write or change the code, and we don't make decisions about the scope of the product. Instead, we help to defend the value that's there by alerting management to important problems that threaten that value. From another perspective, testers may add value by identifying alternative uses for the product—potential purposes or approaches towards using the product that may not have been realized before we started interacting with it. Perhaps the most comprehensive view of the tester's relationship with value is to suggest a different view of what we're testing: if the product is a system—the sum of the code and everything we know about it—then testers add value by adding to the overall knowledge of the system.

W. Ross Ashby coined the Law of Requisite Variety, which suggests that if one system is to control another, the controlling system has to be more complex that the system being controlled. Applied to testing, the Law of Requisite Variety suggests that "if you want to understand something complicated, you must complicate yourself". (G.F. Smith, quoted in Weick's *Sensemaking in Organizations*).

There's a meme afoot in the testing business that suggests that we should make testing easier. Certainly we should make aspects of testing easier, but testing itself isn't supposed to be easy if we're investigating something complex. The simpler and fewer our tests, in general, the simpler and fewer problems that we'll find. So how do we deal with complexity in software and in the uncountable business domains in which it operates?

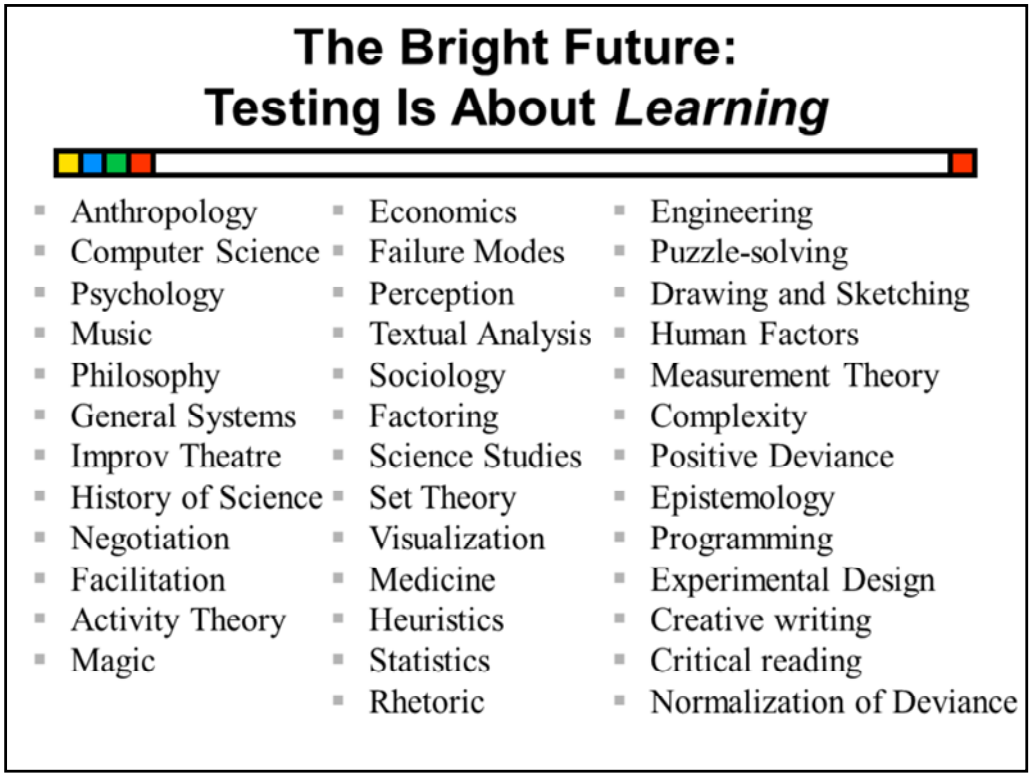
The answer, I believe is to complicate ourselves by broadening our knowledge, experience, and studies.



In the Bright Future, we'll take James Bach's definition of test automation—any use of tools to support testing. We'll recognize that the key word in that definition is not "tools", but "*support*". Automation *assists* the testing effort, but doesn't replace it.

Automation is a medium, in the sense that Marshall McLuhan talked about media. McLuhan defined a medium as anything that causes a change—a tool, a technology, a book, a cup of coffee; he wasn't just referring to communications media, but to all creations of the human mind. Every medium, he said, extends or enhances or intensifies or accelerates some human capability in some way. Automation certainly does that. But McLuhan also pointed out that every medium, when extended beyond its original or intended capacity, reverses into the opposite of its original or intended effect. Cars make it possible to move far faster than we can walk—but when there are too many cars, we have traffic jams, and in many cities at rush hour, it's now faster to walk than to drive. Mobile phones extend our presence to virtually everywhere on the planet, but if we're in conversation with someone whose mobile phone rings, their extended presence results in our own disappearance—that person is no longer with us. Test automation, a medium is subject to the Laws of Media. Automation is an extension of our minds and our capabilities. It doesn't guarantee that we do a better job at testing. If we're doing bad testing, automation will allow us to do more bad testing, faster than ever. Is that what we want?

Machines have all kinds of capabilities. They're excellent for performing high-speed, simple- and single-oracle tasks. They're very reliable, and they don't get tired. They can generate gobs of data, randomize inputs, check for syntactical correctness. With visualization tools, they can assist people in performing high-skill, high-cognition tasks. We can and should use automation tools wherever we can use their help. But we should always remember that the tools are not the testers; we are.



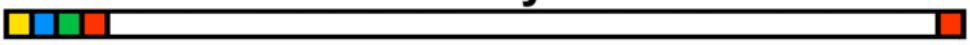
A tester once told me about a problem that his company encountered with a police dispatching system that was designed to be used in a number of cities. The designers and owners of the product believed that everything was okay, and a pilot deployment appeared to go well. However, each city had somewhat different policies and practices. Although the system worked well for the first city, other cities reported serious problems.

The company addressed the issue by sending testers out to observe actual users of the system. At first the testers merely watched, and said very little. Over a few days, the testers were able to refine their observations and their questions for the users, and were even able to observe problems that the users themselves didn't quite notice. This is similar to the practice of participant observation, a key aspect of the social science of anthropology.

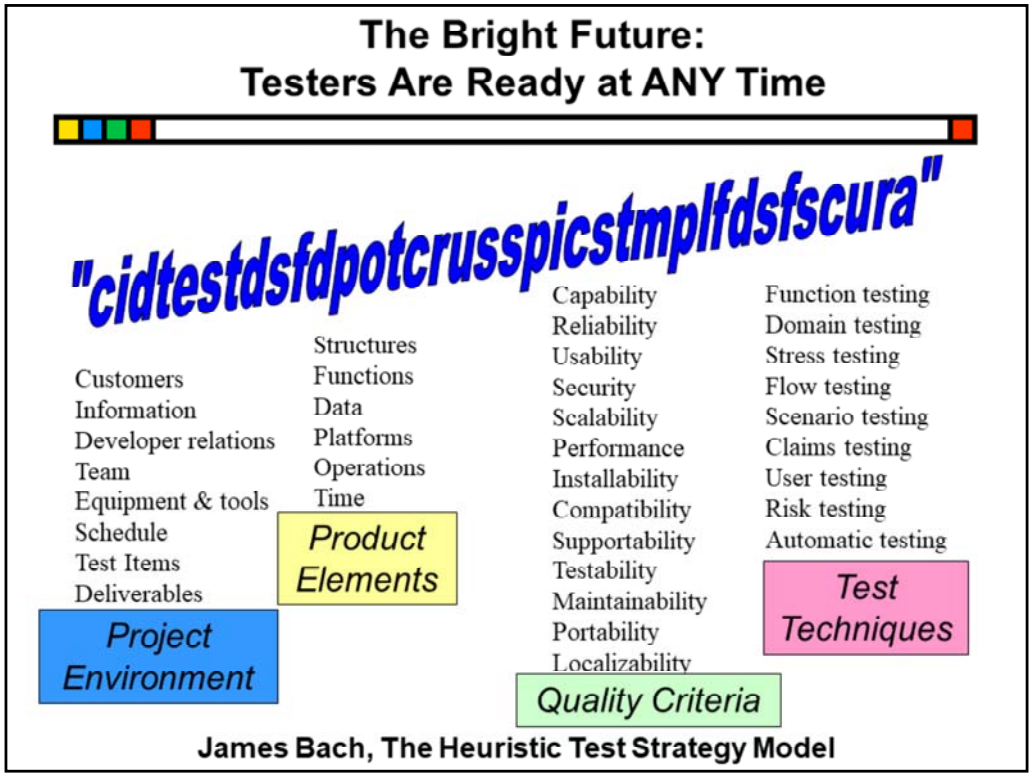
Over the last couple of years, I've read a number of books about economics, about neurology, about the way we process emotions, about evolution, about medicine, about general systems, about psychology, about heuristics. None of them are books about testing… yet *all* of them are books about testing. Each has something of great value that the testing community could put to work.

Perhaps Cem Kaner says it best in his talks about testing as a social science. Like the social sciences, excellent testing is about determining the impact of a software product on people; it involves work with both quantitative and qualitative research; it requires tolerance for ambiguity, partial answers, and situationally specific results; issues related to ethics and values are relevant; diversity of values and interpretations is normal; and observer bias is a fact of life and is well-managed. Testing cannot provide us with complete answers. But testing does provide us with partial answers that might be useful. In the Bright Future, testers will study and use approaches from the social sciences, in addition to those from computer science, engineering, and mathematics.

The Bright Future: Testers Are Ready at ANY Time



- **History** — The present version of the system *is consistent* with past versions.
- **Image** — The system *is consistent* with an image that the organization wants to project.
- **Comparable Products** — The system *is consistent* with systems that are similar or comparable in some way.
- **Claims** — The system *is consistent* with what important people say it's supposed to be.
- **User Expectations** — The system *is consistent* with what users could reasonably want.
- **Product** — Each element in the system *is consistent* with comparable elements in the same system.
- **Purpose** — The system *is consistent* with its purposes, both explicit and implicit.
- **Statutes** — The system *is consistent* with applicable laws or relevant standards.



See a detailed explanation of the Heuristic Test Strategy Model on the Satisfice Web site at http://www.satisfice.com. The ideal thing is to carry them around in your head. Here's the five-day plan:

Day 1: Repeat several times the list "Project environment", "Product Elements", "Quality Criteria", and "Test Techniques". If it helps, sketch them in the form that you see on page 14. Say the mnemonics CIDTEST, San Francisco Depot, CRUSSPIC STMPL, and FeDSFeSCURA out loud a few times. Remember how silly they sound.

Day 2: Learn the Project Environment: "CIDTESTD – Mother Approved", and the guidewords that go with it. Take the questions from the list on page 5 of the appendices to this course (later in this book), and ask those questions about your project.

Day 3: Learn Product Elements: "San Francisco DePOT", and the associated guidewords. See page 6 of the appendices; model the product based on the expanded list there.

Day 4 (the tough day): Learn the Quality Criteria, "CRUSSPIC STMPL", and the guidewords. Remember that all of the letters of Crusspic's name stand for an "-ility", except for Security and Performance. See page 7 of the appendices, and think about the ways that these criteria relate to your product, its users, and your clients.

Day 5 (a little easier, plus you're done!): Test Techniques, "FDSFSCURA". Learn the names of the techniques. See page 4 of the appendices, and try a new approach on your product.

Many of the points in this model are described in Michael's articles for Better Software Magazine at http://www.developsense.com/articles.

We've found this to be an excellent model for generating test ideas, and using it is a fine testing skill. An even more important testing skill is the ability to generate new models of guideword heuristics based on your context and needs. Consider Mike Kelly's guideword heuristics for touring a product—"FCC CUTS VIDS"--or his MCOASTER model, or Jonathan Kohl's FP DICTUMM. (Google for them!)

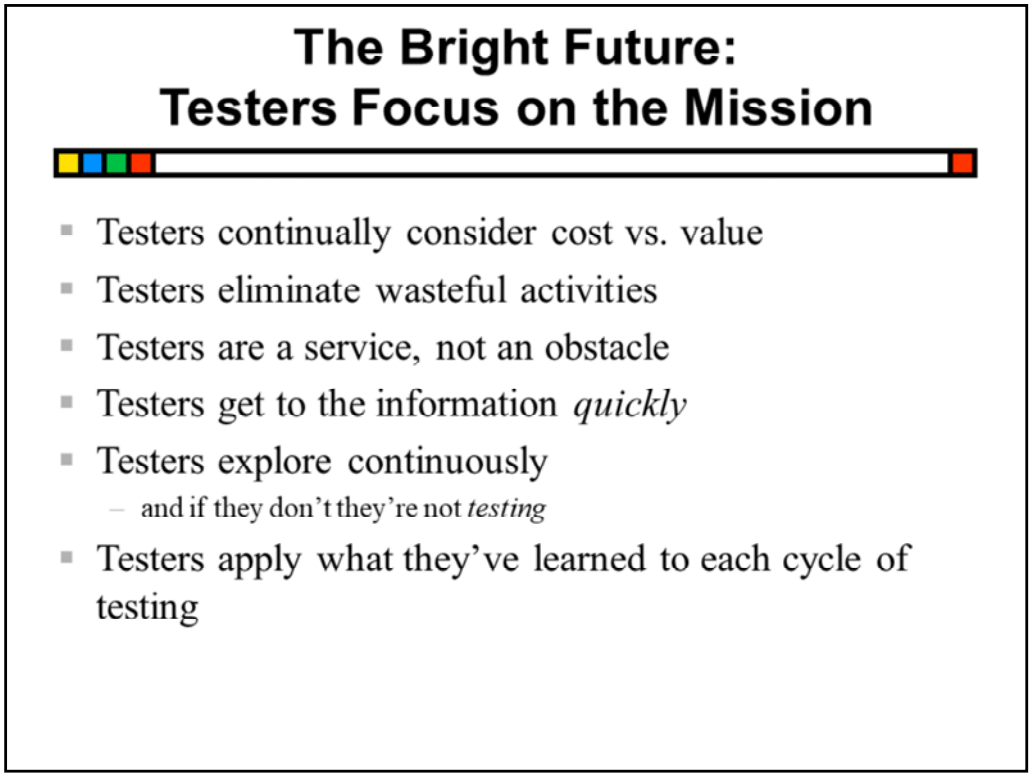
Notice Anything About The Flowers?
(Please answer silently!)

What About When We Change the Question?

Please keep your answers to yourself!

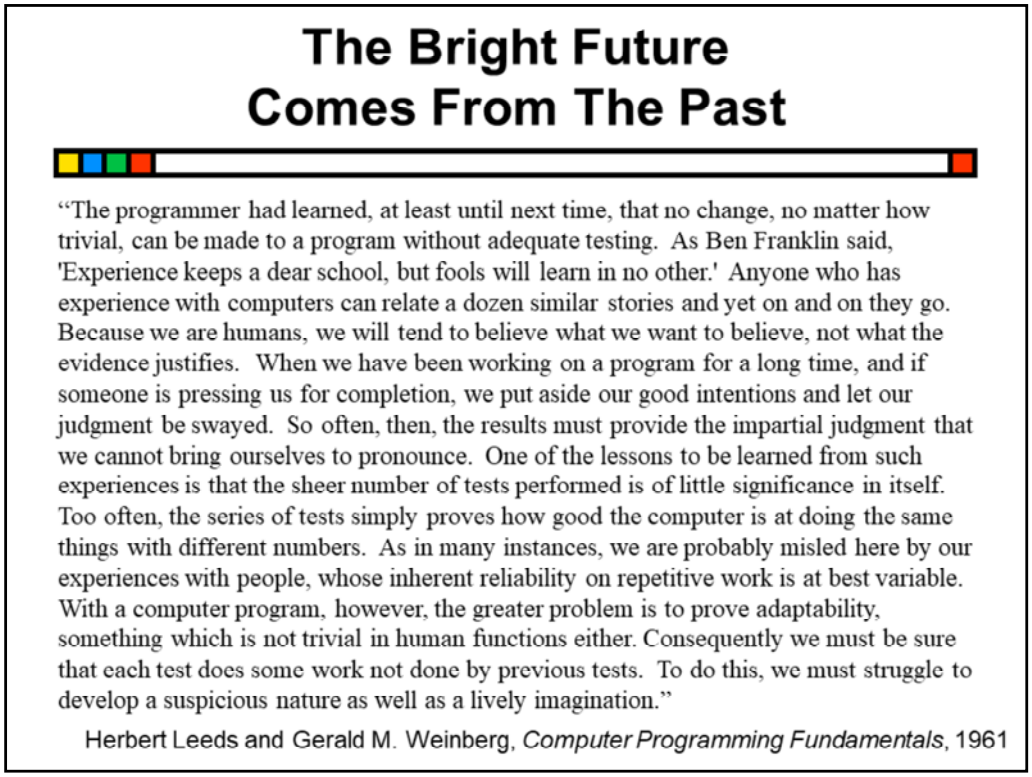
- Notice anything when you look *between* the flowers?
- What happens when we treat the flowers as *ground* and treat the background as *figure*?

Now you can answer out loud!



In 2006, the participants at the Workshop on Heuristic and Exploratory Techniques came up with this definition of exploratory testing under the curatorship of Cem Kaner: "Exploratory testing is a style of testing that emphasizes the freedom and responsibility of the individual tester to continually optimize the quality of his or her work by treating test design, test execution, test result interpretation, and learning as activities that continue in parallel throughout the project." Continual optimization is a key component of this definition, and a key activity in excellent testing.

Rather than preparing mounds of overly detailed plans, excellent testers consider alternative strategies. Might it be more valuable to investigate the product or its artifacts for a while first, prepare lightweight planning documents, and then test, placing the emphasis on test results rather than test plans? Rather than treating high-level automated tests as an assumed good, excellent testers consider development, maintenance, and opportunity costs balanced against the value that the automated tests provide. For each test or test cycle, excellent testers try to reduce the costs of their work, and actively decide which is the most valuable test to perform right now.



James Bach introduced me to this lovely passage. It was written in 1961, the topsy-turvy year in which I was born. We've learned so much, and so little, since then. For me, as it emphasizes adaptability, skepticism, and imagination, this passage tells us where we should be headed in the Bright Future.

Testers Light The Way

Testing is NOT about the mythodology.

We question testing folklore.

We think critically about our own work.

We question our context and our choices, both of which evolve over time.

Who I Am

Michael Bolton

(not the singer, not the guy in Office Space)

DevelopSense, Toronto, Canada

mb@developsense.com

+1 (416) 992-8378

http://www.developsense.com

- *I teach Rapid Software Testing*
  - *http://www.developsense.com/courses.html*
- *I help people to solve testing problems*
  - *http://www.developsense.com/consulting.html*

Acknowledgements

- James Bach
  - some of the material comes from the Rapid Software Testing Course, of which James is the senior author and I am co-author
- Cem Kaner
  - many of the ideas in this talk
- Bret Pettichord
- Jerry Weinberg

Bibliography
How To Think About Testing

- *Perfect Software and Other Illusions About Testing*
  - Gerald M. Weinberg
- *Lessons Learned in Software Testing*
  - Cem Kaner, James Bach, and Bret Pettichord
- "Software Testing as a Social Science"
  - Cem Kaner; http://www.kaner.com/pdfs/KanerSocialScienceSTEP.pdf
- *Testing Computer Software*
  - Cem Kaner, Jack Falk, and Hung Quoc Nguyen
- *An Introduction to General Systems Thinking*
  - Gerald M. Weinberg
- *Exploring Requirements: Quality Before Design*
  - Gerald M. Weinberg

Bibliography
Test Techniques

- *A Practitioner's Guide to Test Design*
  - Lee Copeland
- *How to Break Software*
  - James Whittaker
- *How to Break Software Security*
  - James Whittaker and Herbert Thompson
- *Lessons Learned in Software Testing*
  - Cem Kaner, James Bach, and Bret Pettichord
- *Testing Applications on the Web*
  - Hung Quoc Nguyuen
- *Hacking Web Applications Exposed*
  - Joel Scambray and Mike Shema

Bibliography
Jerry Weinberg

- *Quality Software Management Vol. 1: Systems Thinking*
- *Quality Software Management Vol. 2: First Order Measurement*
- *Secrets of Consulting: How to Give and Get Advice Successfully*
- **Anything** by Jerry Weinberg

Bibliography
Richard Feynman

- *The Pleasure of Finding Things Out*
  - see the Appendix to the Challenger Report.
- *Surely You're Joking, Dr. Feynman! Adventures of a Curious Character*
- *What Do You Care About What Other People Think?*

Bibliography
Outside the Discipline

- *The Social Life of Information*
  - Paul Duguid and John Seely Brown
- *Please Understand Me*
  - David Kiersey
  - The Myers-Briggs Type Inventory, which provides insight into your own preferences and why *other people* seem to think so strangely
- *The Visual Display of Quantitative Information*
  - Edward Tufte
  - How to present information in persuasive, compelling, and beautiful ways
- *A Pattern Language*
  - Christopher Alexander et. al
  - A book about architecture
  - even more interesting as a book about thinking and creating similar but unique things—like computer programs and tests for them

Web Resources

- Michael Bolton http://www.developsense.com
- James Bach http://www.satisfice.com
- Cem Kaner http://www.kaner.com
- The Florida Institute of Technology
  - http://www.testingeducation.org
  - http://www.testingeducation.org/BBST/index.html
- StickyMinds http://www.StickyMinds.com
- Risks Digest http://catless.ncl.ac.uk/risks