Six Short Talks About Software Testing

Michael Bolton
DevelopSense
http://www.developsense.com
michael@developsense.com
+1 (416) 656-5160

Six Talks About Software Testing

1. Oracles
2. Three Sources For Project Information
3. Emotions and Oracles
4. Test Coverage
5. Confirmation vs. Exploration
6. Some Reasons Why Testing Takes So Long

Acknowledgements

- James Bach
  - senior author of Rapid Software Testing, which he and I teach
- Cem Kaner
- Jerry Weinberg

1. Oracles

What is an oracle?

No, not the database

ORACLE

Not the database

An oracle is...

a principle or mechanism

by which

we recognize a problem

But wait…

How can we be

SURE

that we're seeing a problem?

WE CAN'T.

Certainty isn't available.

But we DO have *heuristics*

Heuristics are fallible, "fast and frugal" methods of solving problems, making decisions, or accomplishing tasks.

"The engineering method is the use of *heuristics* to cause the best change in a poorly understood situation within the available resources."
Billy Vaughan Koen
Discussion of the Method

Heuristics: Generating Solutions Quickly

- **adjective:**
  "serving to discover or learn."
- **noun:**
  "A **fallible** method for **solving a problem** or **making a decision**."

"Heuristic reasoning is not regarded as final and strict but as provisional and plausible only, whose purpose is to discover the solution to the present problem."
- George Polya, *How to Solve It*

Oracles

An oracle is a **heuristic** principle or mechanism by which you recognize a problem or make a decision.

"It works!"

really means…

"…it appeared at least once to meet some requirement to some degree."

One or more successes!

It's (not) OK *according to an oracle*.

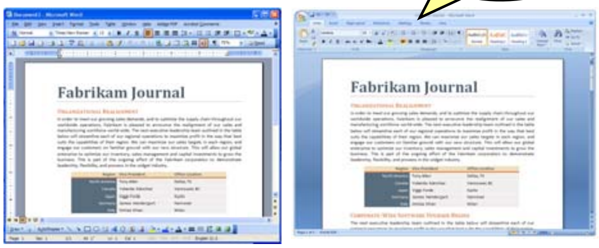
Without an oracle you **cannot** recognize a problem

and conversely…

If you think you see a problem, you **must** be using an oracle… so what is it?

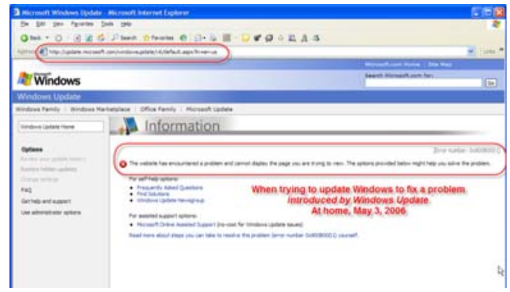
History

Hey, I **liked** the menu bar? How the #&@ do I print **now**?



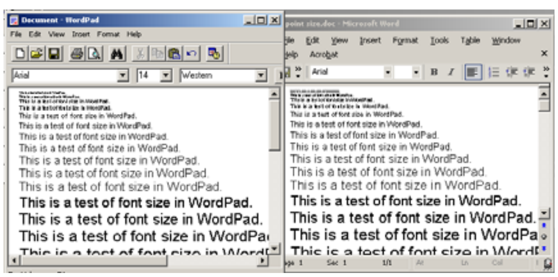
If a product is inconsistent with previous versions of itself, we suspect that there might be a problem.

Image



If a product is inconsistent with an image that the company wants to project, we suspect a problem.

Comparable Products



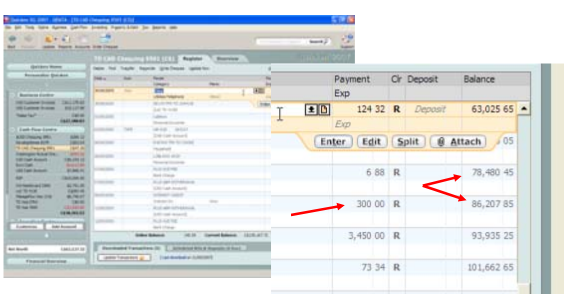
WordPad Word

When a product seems inconsistent with a comparable product, we suspect that there might be a problem.

Claims

When a product is inconsistent with claims that important people make about it, we suspect a problem.

User Expectations

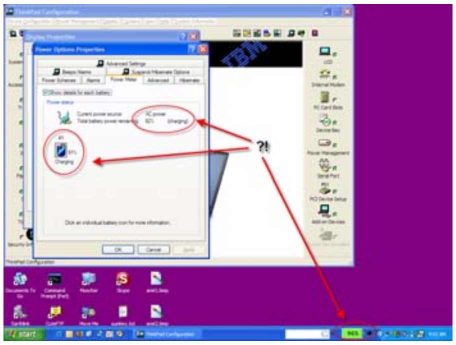


When a product is inconsistent with expectations that a reasonable user might have, we suspect a problem.

Purpose

When a product is inconsistent with its designers' explicit or implicit purposes, we suspect a problem.

Product



When a product is inconsistent internally—as when it contradicts itself—we suspect a problem.

Statutes and Standards

When a product is inconsistent with laws or widely accepted standards, we suspect a problem.

We like consistency when…

- the present version of the system *is consistent* with **past versions** of itself.
- the system *is consistent* with **an image** that the organization wants to project.
- the system *is consistent* with **comparable systems**.
- the system *is consistent* with **what important people say** it's supposed to be.
- the system *is consistent* with **what users seem to want**.
- each element of the system *is consistent* with comparable **elements in the same system**.
- the system *is consistent* with implicit and explicit **purposes**.
- the system *is consistent* with applicable **laws** or **standards**.

unless it's a problem.

- We like it when the system ***is not consistent*** with patterns of familiar problems.

But...

- All of the consistency oracles are heuristic.

Can work.

Might fail.

All Oracles Are Heuristic

- No single oracle can tell us whether a program or feature is working correctly at all times and in all circumstances.
- Oracles are fallible and context-dependent.
- Oracles can be contradicted by other oracles.
- Multiple oracles may increase our confidence, but even combinations of oracles are fallible.
- A problem revealed by a single oracle may devastating.
- Recognizing a different problem usually requires a different oracle.
- A test designer need not be aware of an oracle in advance of the observation, unless the test is designed to be run by rote.

Oracles: Strength in Diversity

An oracle doesn't tell you that there IS a problem. An oracle tells you that you *might be seeing a problem.*

Consistency heuristics rely on the quality of your models of the product and its context.

Rely solely on documented, anticipated sources of oracles, and your testing will likely be weaker.

How Do I Keep Track? HICCUPPS!

- History
- Image
- Comparable Products
- Claims
- User Expectations
- Purpose
- Product
- Statutes (or Standards)

…plus for "Familiar Problems", add that inconsistent F!

Remember…

For skilled testers, good testing isn't just about pass vs. fail.

For skilled testers, testing is about ***problem vs. no problem.***

End of Talk One

2. Requirements

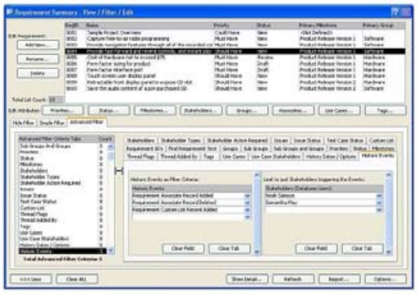
What are requirements?

What *Are* Requirements?

Do your requirements look like this?

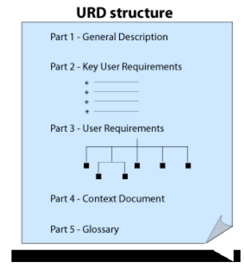
What *Are* Requirements?

Or like this?



What *Are* Requirements?

Or like this?



What *Are* Requirements?

What about this?

Let's talk about *reification*

The error of confusing
concepts and constructs
for
things.

Let's talk about *reification*

The error of confusing
ideas
with their
containers.

Reification

It's like the national sport
of software engineering

What Gets Subjected to Reification?

Randomness Complexity
Boundaries Quality
Ambiguity Purpose
Bugs Test cases
and…

What Gets Subjected to Reification?

REQUIREMENTS

Reification Happens When We…

- …count things that can’t be counted
- …point to things that can’t be pointed to
- …mistake relationships for attributes
- …confuse the container and the contents
- …mistake the map for the territory

NOT The Swiss Alps

The Swiss Alps

The Swiss Alps

~~NOT~~ The Swiss Alps

So, Lesson One

Don't confuse

"the territory"

and

"the map of the territory"

So, Lesson One

Don't confuse

"the requirements"

and

"the requirements document"

So where *do* we get information?

- Yes, from documents
  - marketing requirements documents
  - functional specifications
  - standards, such as RFCs
  - note: documents often point to other documents
  - help files
  - user manuals
- Prototypes
- Previous products
- Test programs

REFERENCE

So where *do* we get information?

- Product experience
- Domain and customer knowledge
- Testing or support knowledge
- Platform experience
- General knowledge
- Affordance
- (In)consistency heuristics
  - about which more later

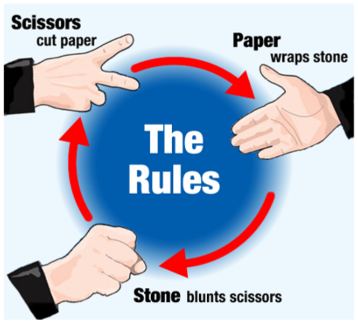
INFERENCE

So where do we get information?

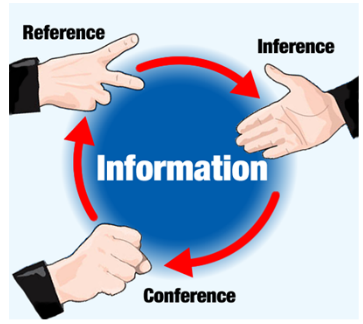
- One-on-one meetings
- Design meetings
- Scrums
- Bug triage meetings
- Technical support information
- Email threads
- Water-cooler conversations
- Explicit decisions from project management

CONFERENCE

Project Information is Often *Discovered* and *Refined*



Project Information is Often *Discovered* and *Refined*



To Enrich Testing

- Accept that context and choices change over time as we learn
- Consider all the different forms of information that are available to you
- Use your inferences, and practice describing how you arrive at them
- Compare and contrast what different people say in different for(u)ms
- Report inconsistencies that you observe
- …then *let management manage the project*

End of Talk Two

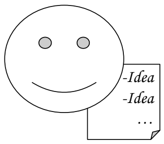
3. Emotions and Oracles

Rapid Software Testing

To be able to test a product when it has to be tested *right now*, under *conditions of uncertainty*, in a way that stands up to *scrutiny*.

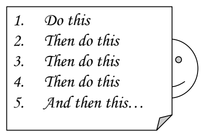
Skill + Heuristics makes testing powerful

This…



The skilled tester remains in control of the process.

…not this.



Scripted procedures give the *illusion* of control over unskilled testers.

Most conventional wisdom about testing is empty folklore (yes, Agile Testing too)

Oracles

An *oracle* is a heuristic principle or mechanism by which someone might recognize a problem.

(usually works, might fail)

(but not decide conclusively)

📖 Bug (n): Something that bugs someone who matters

Oracles

- When something is okay, it's because an oracle says so.
- When something is wrong, it's with respect to some oracle.

You're blind to a problem if you don't have an oracle for it.

Consistency ("this agrees with that")

an important theme in oracles

History
Image
Comparable Products
Claims
User Expectations
Purpose
Product
Standards

Consistency heuristics rely on the quality of your models of the product and its context.

Noticing Problems

- We run the program, and to some, it seems like we just notice problems as we go along.
- Noticing problems is a very logical, objective, dispassionate process.

Isn't it?

One Way of Thinking of Testing

Give me only *programmers* to test my code!

Hey… how come we have 650 open reqs for SDET*s?

And how come everybody gripes about security and usability?

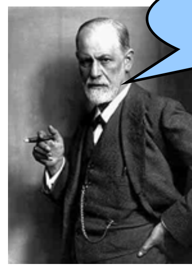
* SDET = "Software Development Engineer in Test"

Besides… Automation Can't…

reframe refine investigate speculate
empathize anticipate predict suggest
judge project
recognize refocus
appreciate conceptualize elaborate
become resigned strategize evaluate
teach question charter assess
learn get frustrated
work around a problem
invent
model make conscious decisions
resource
troubleshoot collaborate

THINK

Machines Don't Get Aroused



arousal (n.): a physiological and psychological state of being awake.

important in regulating consciousness, attention, and information processing.

Machines are cool…

- …but they don't get aroused.
- That is, they don't notice problems…
- …and they can't even try.

Skilled testers don't think "pass or fail"?

Skilled testers ask "is there a *problem* here"?

Machines and automated tests don't even know to ask.

Examples of Common Cognitive Biases

- Fundamental Attribution Error
  - "THIS is what that is, and that's all that it is."
- Anchoring Bias (overcommitting to an idea)
  - "I don't have to reconsider."
- Automation Bias (machines over people)
  - "A machine told me; it must be true."
- Reification Error (counting the uncountable)
  - "How many ideas did you have today?"

Using Emotion To Help Overcome Bias

- Your biases may cause you to miss bugs
- An emotional reaction is a trigger to learning
- Without emotion, we don't reason well
  - check the psych literature
- When you find yourself mildly concerned about something, someone else could be *very* concerned about it

An emotion is a signal; consider looking into it

Emotional Triggers

What might feelings be telling us?

- Impatience ⇨ an intolerable delay?
- Frustration ⇨ a poorly-conceived workflow?
- Amusement ⇨ a threat to someone's image?
- Surprise ⇨ inconsistency with expectations?
- Confusion ⇨ unclear interface? poor testability?
- Annoyance ⇨ a missing feature?
- Boredom ⇨ an uninteresting test?
- Tiredness ⇨ time for a break?

Our clients are human

- *Our* humanity as testers helps to reveal important information about our products.
- Emotions provide a rich source of oracles—principles or mechanisms by which we recognize problems.
- I'll wager that any time we've seen a bug, our emotions were a big factor in recognizing or interpreting it.
- Why do so many in our profession seem to be so oblivious to the value of emotions?

End of Talk Three

4. Congruence Bias

The biggest problem in testing?

I teach people how to test software.

James Bach and I co-author and teach a course called Rapid Software Testing.

Why do we bother to teach such an unimportant, trivial subject?

It's easy to trivialize something if you haven't thought much about it.

At first, the testing task *does* sound pretty trivial.

"Try it and see if it works."

Actually, that is a pretty trivial task.

Most programs, even really shoddy ones, can do something, once.

So if we simply treat testing as the task of showing that the program can do something, once, then testing is trivial.

But demonstrating that something works isn't usually why we test.

Some people say that we test to assure quality.

But that doesn't make sense.

We 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,
do that stuff.

Quality is value to
some person(s)
who matter.

Quality is not
something in the product.

Quality is a relationship
between the product and
some person who matters.

Managers get to decide
who matters.

Decisions about quality
are political decisions.

Managers have the authority
to make business decisions.

So managers, not testers
are the real
quality assurance people.

Testers don't
manage the project.

We're not the brain.

We're the antennae.

We think observe and think critically about software, and report what we observe.

We don't make the decisions; we provide information to decision-makers.

Some people say we test to make sure the product fulfills its requirements.

Yet "requirements" is like "quality".

A requirement isn't a thing in itself.

A requirement
a difference between
what we've got and
what someone wants.

That is, a requirement is
a relationship
between the product and the
person who wants it.

Presumably, they want it for
some purpose.

But like "quality" and
"requirement",
purpose is a relationship.

Different people,
different purposes.

If a product fulfills its purpose,
we often say that
"the product works".

So maybe we test to make sure that it works.

"It works",
according to Jerry Weinberg,
are the two most ambiguous
words in the English language.

"It works",
according to Jerry Weinberg,
are the two most ambiguous
words in the English language.

"It"
can mean
anything.

"works"
means
seems to do something.

"It works"
really means
"it appears to do something…"

"…to meet some person's requirements…"

"…to some degree…"

"…in some circumstance…"

"…at some time…"

"…at least on my machine."

Product development is an essentially optimistic activity.

If we weren't
sufficiently optimistic,
we wouldn't bother trying.

On the other hand,
when we're optimistic,
we tend to forget something.

If the product
can work for some person,
it might fail for that person.

Forgetting to think about
how something might fail
(or forgetting to test for it)
is a problem with a name:

Congruence bias.

Congruence bias
is sometimes known as
confirmation bias.

Congruence bias
indulges our desire
to write a few simple tests
that are likely to pass.

Congruence bias also
suppresses our desire to
perform tests that might fail.

So the mission of testing
is subtly complex
—something like…

Try it to learn sufficiently…

…everything that matters…

…about how it can work…

…and how it might fail.

Let’s look at that again.

Try it to learn sufficiently
everything that matters
about how it can work
and how it might fail.

“Try it” means
to configure,
operate,
observe,
and evaluate it.

“To learn” means
to discover stuff
that we didn’t know before,
or that we weren’t sure about.

“Sufficiently”
does double duty—
“*try it*” sufficiently, and
“*to learn*” sufficiently.

"Everything that matters"
is also important
from two angles.

"Everything that matters"
both expands and contracts
our scope.

We don't have a lot of time
to test stuff
that doesn't matter.

And, by the same token,
we don't want to miss testing
stuff that does matter.

"How it can work"
is not really *that* big a deal,
because we can demonstrate that
"it works"
at least once.

Choices about
the tests that we run (or not)
are governed by
our mindset.

The programmer tends to need to run tests that confirm that the product still works.

In fact, programmers can even design automated unit or acceptance tests to show that it works.

Automating lower-level tests—
change detectors,
as Cem Kaner calls them—
is a fine thing for developers to do.

And, in fact, it might even be a good idea to write some of those tests before we write some code.

This becomes a dangerous business for a tester, though.

Why?

We will discover and learn
many things
as we develop the product.

We risk
wasting time and effort
when we write too many tests
(or test cases) too early
for a product that
we don't yet understand.

So, it might be a good idea
to start with a few tests now,
and add more later.

We add more tests
as we learn more
about what we value, and
what might threaten that value.

Shouldn't we also drop tests as we
learn more about things that pose
lesser threats?

To be effective, the tester
needs a different mindset from
the optimistic developers and
product managers.

Testers need mostly to run tests that attempt to demonstrate that the product might fail.

If the product works despite the challenge, then we get a free demonstration that the product can work.

But when a tester performs a confirmatory test, she misses an opportunity to perform an investigative test.

When we don't vary our tests...

When we don't apply what we've learned...

When we don't run towards the risk...

When we don't
question our beliefs...

We run the risk of
missing serious problems
that threaten
the value of the product.

This risk is itself
a serious problem
that threatens
the value of the business.

How do we solve it?

By focusing on this:

Testing is not merely
verification.

Testing is not merely validation.

Testing is not merely confirmation.

Testing is much more importantly about exploration.

Testing is much more importantly about discovery.

Testing is much more importantly about investigation.

Testing is much more importantly about learning.

Exploration,
not confirmation,

helps to defend our clients
(and ourselves)
from congruence bias

and that helps in one of our
primary jobs:

defending value in a product.

Plus one more cool thing.

Remember that stuff about
different people,
different purposes?

Exploration can help us to identify new people and infer new purposes.

Finding a new purpose means revealing new value for a product.

This affords testers the chance not only to defend value, but to add it.

Congruence bias is a big problem in software development.

Thinking in terms of skilled testing...

Thinking in terms of exploration, discovery, investigation, and learning poses a big solution.

End of Talk Four

5. Test Coverage

What IS Coverage?

Coverage is "how much of the product we have tested."

It's the extent to which we have traveled over *some map* of the product.

...but what does it mean to "map" a product? Talking about coverage means talking about

models

Models

- **A model is a heuristic idea, activity, or object…**

  such as an *idea in your mind*, a *diagram*, a *list of words*, a *spreadsheet*, a *person*, a *toy*, an *equation*, a *demonstration*, or a *program*

- **…that represents (literally, *re*-presents) another idea, activity, or object…**

  such as something complex that you need to work with or study

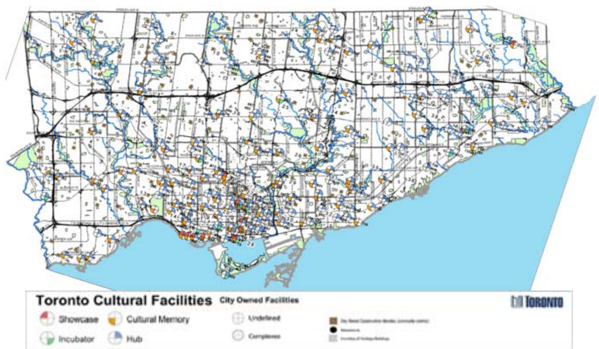
- **…whereby understanding something about the model may help you to understand or manipulate the thing that it represents.**

- A *map* is a model that helps to navigate across a terrain.
- *2+2=4* is a model for adding two apples to a basket that already has two apples.
- *Atmospheric models* help predict where hurricanes will go.
- A *fashion model* helps understand how clothing would look on actual humans.
- *Your beliefs about what you test are a model of what you test.*

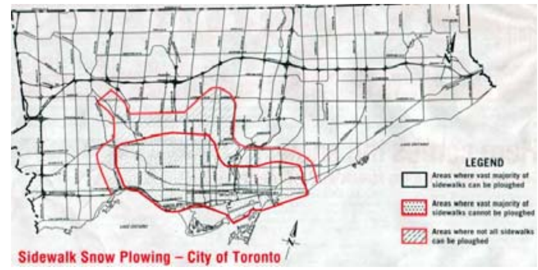
A Map of the Toronto Subway

Here's Another One

A Map of Toronto's Cultural Facilities



So You Want Your Sidewalk Plowed?



A Bike Ride?

What Is Covered Incidentally?

Different Maps Show Different Things

- The information that we care about may be incidental to the "purpose" of the map

It depends on what we're looking for...

It depends on where we're looking...

It depends on what it means to "cover" the map!

There are as many kinds of test coverage as there are ways to model the system.

Structure Business Risk Time Technical Risk Functions Platform Data Operations

And each could be...

Intentional or... Accidental

One Way to Model Coverage: Product Elements (with Quality Criteria)

SFDPOT -- San Francisco Depot

Product Elements

- Structure
- Function
- Data
- Platform
- Operations
- Time

General Focusing Heuristics

- use test-first approach or unit testing for better *code* coverage
- work from prepared test coverage outlines and risk lists
- use diagrams, state models, and the like, and cover them
- apply specific test techniques to address particular coverage areas
- make careful observations and match to expectations

Follow your procedures.

To do this *more rapidly*, make *preparation* and *artifacts* fast and frugal: leverage existing materials and avoid repeating yourself. Emphasize doing; relax planning. You'll make discoveries along the way!

General Defocusing Heuristics

- diversify your models; intentional coverage in one area can lead to unintentional coverage in other areas—this is a Good Thing
- diversify your test techniques
- be alert to problems other than the ones that you're actively looking for
- welcome and embrace distraction
- do some testing that is *not* oriented towards a specific risk
- use high-volume, randomized automated tests

Question and vary your procedures.

How Might We Organize, Record, and Report Coverage?

- automated tools (e.g. profilers, coverage tools)
- annotated diagrams (as shown in earlier slides)
- coverage matrices
- bug taxonomies
- Michael Hunter's You Are Not Done Yet list
- James Bach's Heuristic Test Strategy Model
  - described at www.satisfice.com
  - articles about it at www.developsense.com
- Mike Kelly's MCOASTER model
- coverage outlines and risk lists
- session-based test management

Wait! What About Quantifying Coverage?

- A nice idea, but we don't know how to do it in a way that is consistent with *basic* measurement theory
  1. If we describe coverage by counting test cases, we're committing reification error.
  2. If we use percentages to quantify coverage, we need to establish what 100% looks like.
  3. Complex systems may display emergent behaviour.

How do we hang meaningful numbers on that stuff?

Extent of Coverage

- Smoke and sanity
  - Can this thing even be tested at all?
- Common and critical
  - Can this thing do the things it *must* do?
  - Does it handle happy paths and regular input?
  - *Can* it work?
- Complex, extreme and exceptional
  - Will this thing handle challenging tests, complex data flows, and malformed input, etc.?
  - *Will* it work?

End of Talk Five

6. When Do We Stop Testing?

If you're a tester, you've been asked…

Why is testing taking so long?

or…

When are you going to be done testing?

…and if you *haven't* been asked, just stick around for a while.

But Hold On A Sec…

Are we ever done testing?

- There are always more conditions to check
- There are always more operations to perform
- There are always more platforms to set up
- There are always more variations of timing to try

Testing Doesn't Stop On Its Own

We decide to stop testing.

Whether for a particular test, a given test cycle, or a ~~test~~ development project, we stop testing *when we decide* to stop testing.

The fact is…

Testing is done when management decides to ship the product.

The decision to ship a product

IS NOT…	IS…
• made by the testers	• made by the client
• governed by rules	• governed by heuristics
• a technical decision	• a business decision
• based on whether **testing** is finished	• based on whether **development** is finished

Testing doesn't make the decision
Testing helps to inform the decision

Another fact…

Management decides to ship the product when development is done.

So the real question is…

Why is development taking so long?

Isn't that a question for the whole team?

Test Session Effectiveness

- A "perfectly effective" testing session is one entirely dedicated to test design, test execution, and learning
  - a "perfect" session is the exception, not the rule
- Test design and execution tend to contribute to test coverage
  - varied tests tend to provide more coverage than repeated tests
- Setup, bug investigation, and reporting take time away from test design and execution

Modeling Test Effort

Suppose that testing a feature takes two minutes

- this is a highly arbitrary and artificial assumption—that is, it's *wrong*, but we use it to model an issue and make a point

- Suppose also that it takes an extra eight minutes to investigate and report a bug that we found with a test
  - another stupid, sweeping generalization in service of the point
- In a 90-minute session, we can run 45 feature tests—*as long as we don't find any bugs*

How Do We Spend Time?
(assuming all tests below are *good* tests)

Module	Bug reporting/investigation (time spent on tests that find bugs)	Test design and execution (time spent on tests that find no bugs)	Number of tests
A (good)	0 minutes (no bugs found)	90 minutes (45 tests)	45
B (okay)	10 minutes (1 bug, 1 test)	80 minutes (40 tests)	41
C (bad)	80 minutes (8 bugs, 8 tests)	10 minutes (5 tests)	13

Investigating and reporting bugs means….

SLOWER TESTING or…

REDUCED COVERAGE …or both.

- In the first instance, our *coverage* is great—but if we're being assessed on the number of bugs we're finding, we look bad.
- In the second instance, coverage looks good, and we found a bug, too.
- In the third instance, we look good because we're finding and reporting lots of *bugs*—but our *coverage* is suffering severely. A system that rewards us or increases confidence based on the number of bugs we find might mislead us into believing that our product is well tested.

What Happens The Next Day?
(assume 6 minutes per bug fix verification)

Fix verifications	Bug reporting and investigation today	Test design and execution today	New tests today	Total over two days
0 min	0	45	45	90
6 min	10 min (1 new bug)	74 min (37 tests)	38	79
48 min	40 min (4 new bugs)	2 min (1 test)	5	18

Finding bugs today means….

VERIFYING FIXES LATER

…which means….

EVEN SLOWER TESTING or…

EVEN LESS COVERAGE …or both.

•…and note the optimistic assumption that all of our fixed verifications worked, and that we found no new bugs while running them. Has this ever happened for you?

With a more buggy product

- More time is spent on bug investigation and reporting
- More time is spent on fix verification
- Less time is available for coverage

Not only do we do more work…

…we also know less about the system

With a *less* buggy product…
(that is, one that has had some level of testing already)

- We've got *some* bugs out of the way already
- *Those* bugs won't require investigation and reporting
- *Those* bugs won't block our ability to test more deeply

So, programmers, please consider this heuristic:

Test early, and test often!

Test Early and Often!

- Recurrent themes in agile development (note the small A)
  - test-first programming
  - automated unit tests, builds, and continuous integration
  - testability hooks in the code
  - lots of customer involvement
- The ideas are
  - to increase developers' confidence in and commitment to what they're providing ("at least it does *this*")
  - to allow rapid feedback when it *doesn't* do *this*
  - to permit robust refactoring
  - to increase test coverage and/or reduce testing time

Test a product as you build it!

Testing vs. Investigation

- Note that I just gave you a compelling-looking table, using simple measures, but notice that we still don't really know anything about…
  - the quality and relevance of the tests
  - the quality and relevance of the bug reports
  - the skill of the testers in finding and reporting bugs
  - the complexity of the respective modules
  - luck

…but if we *ask better questions*, instead of letting data make our decisions, we're more likely to *learn important things*.

To ask better questions…
To answer them more quickly…
To find bugs more easily…
To fix found bugs faster…
Focus on testability!

We Testers Humbly Request…
(from the developers)

- Developer tests at the unit level
  - use TDD, test-first, automated unit tests, reviews and inspections, step through code in the debugger—whatever increases your own confidence that the code does what you think it does

A less buggy product

takes less time to test.

We Testers Humbly Request…
(from the whole team)

- Focus on testability
  - log files
  - scriptable interfaces
  - real-time monitoring capabilities
  - installability and configurability
  - test tools, and help building our own
  - access to "live oracles" and other forms of information

Speed up the decision:

"Problem or no problem?"

End of Talk Six