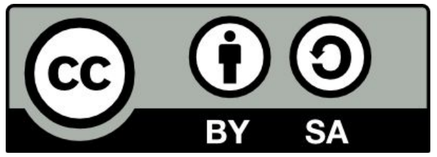
Black Box Software Bug Advocacy
Lecture 1
Basic Concepts

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Rebecca L. Fiedler, M.B.A., PH.D.

Retired, President of Kaner, Fiedler & Associates



Notice

The practices recommended and discussed in this course are useful for an introduction to testing, but more experienced testers will adopt additional practices. I am writing this course with the mass-market software development industry in mind. Mission-critical and life-critical software development efforts involve specific and rigorous procedures that are not described in this course.

Some of the BBST-series courses include some legal information, but you are not my legal client. I do not provide legal advice in the notes or in the course. If you ask a BBST instructor a question about a specific situation, the instructor might use your question as a teaching tool, and answer it in a way that s/he believes would “normally” be true but such an answer may be inappropriate for your particular situation or incorrect in your jurisdiction. Neither I nor any instructor in the BBST series can accept any responsibility for actions that you might take in response to comments about the law made in this course. If you need legal advice, please consult your own attorney.

Many Thanks…

The BBST lectures evolved out of courses co-authored by Kaner & Hung Quoc Nguyen and by Kaner & Doug Hoffman. We then co-taught and evolved the course with James Bach and Michael Bolton and we co-taught and evolved it again with Altom and its instructors. The online adaptation of BBST was designed primarily by Rebecca L. Fiedler.

After being developed by practitioners, the course evolved through academic teaching and research largely funded by the National Science Foundation. The Association for Software Testing served as our learning lab for practitioner courses. We also evolved the 4-week structure with AST. We could not have created this series without AST's collaboration. Since 2014, Altom has been offering the course commercially. Starting with 2019, Altom has been maintaining and updating the course materials.

Many Thanks…

We also thank Jon Bach, Scott Barber, Bernie Berger, Ajay Bhagwat, Rex Black, Jack Falk, Elizabeth Hendrickson, Kathy Iberle, Bob Johnson, Karen Johnson, Brian Lawrence, Brian Marick, John McConda, Melora Svoboda, dozens of participants in the Los Altos Workshops on Software Testing, the Software Test Managers' Roundtable, the Workshops on Heuristic & Exploratory Techniques, the Workshops on Teaching Software Testing, the Austin Workshops on Test Automation and the Toronto Workshops on Software Testing and students in AST courses for critically reviewing materials from the perspective of experienced practitioners.

We also thank the many students and co-instructors at Florida Tech, who helped us evolve the academic versions of this course, especially Pushpa Bhallamudi, Walter P. Bond, Tim Coulter, Sabrina Fay, Ajay Jha, Alan Jorgenson, Kishore Kattamuri, Pat McGee, Sowmya Padmanabhan, Andy Tinkham, and Giri Vijayaraghavan.

We also thank all instructors, practitioners and Altom employees who contribute to updating and developing new content for this course series, especially Ancuța Bodnărescu, Alexandra Casapu, Oana Casapu, Ru Cindrea, Gabriel Dobrițescu, Zoltán Molnár, Ray Oei, and Dolores Pente.

Course Overview: Fundamental Topics

BBST ®
BUG ADVOCACY

1. Basic Concepts 
2. Effective Advocacy: Making People Want to Fix the Bug
3. Writing Clear Bug Reports
4. Anticipating and Dealing With Objections: Irreproducible Bugs
5. Bugs that Could Be Dismissed as Unreasonable, Unrealistic, or Unimportant
6. Credibility and Influence

The Bug Advocacy Lectures

Five Key Challenges

How should we:

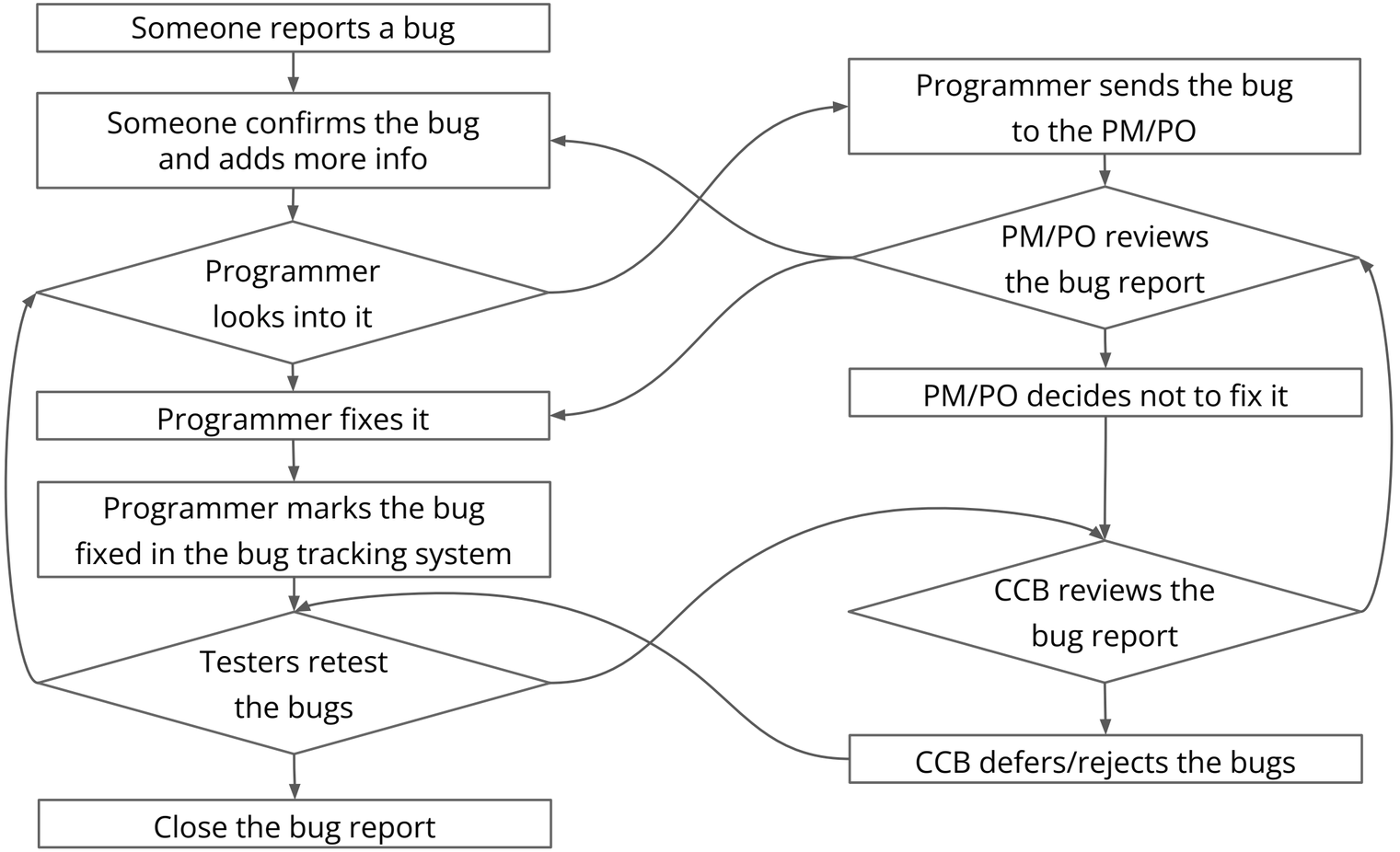
1. Balance time spent:
    - improving our understanding and communication of our findings

    versus

    - finding new bugs?
2. Deal with conflicts of interest among stakeholders?
3. Present problems to people under stress?
4. Preserve our credibility?
5. Preserve our integrity?

Summary of a Bug Workflow

How Workgroups Handle Bug Reports



Summary of a Bug Workflow

Someone reports a bug

- A tester finds a bug, investigates it, and reports it
- Non-testers report bugs too.

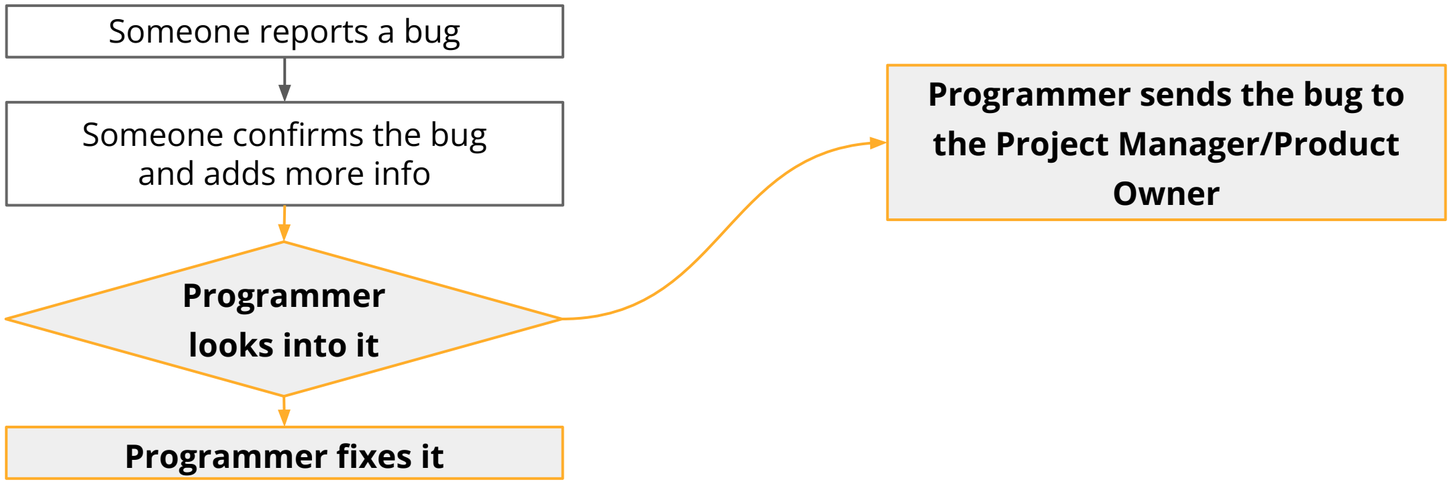
Summary of a Bug Workflow

Someone replicates the bug, to confirm that

- it is a real problem
- the steps in the report accurately describe how to reproduce it.

Confirmation is essential for bugs reported by non-testers, because those reports are often incomplete.

Summary of a Bug Workflow

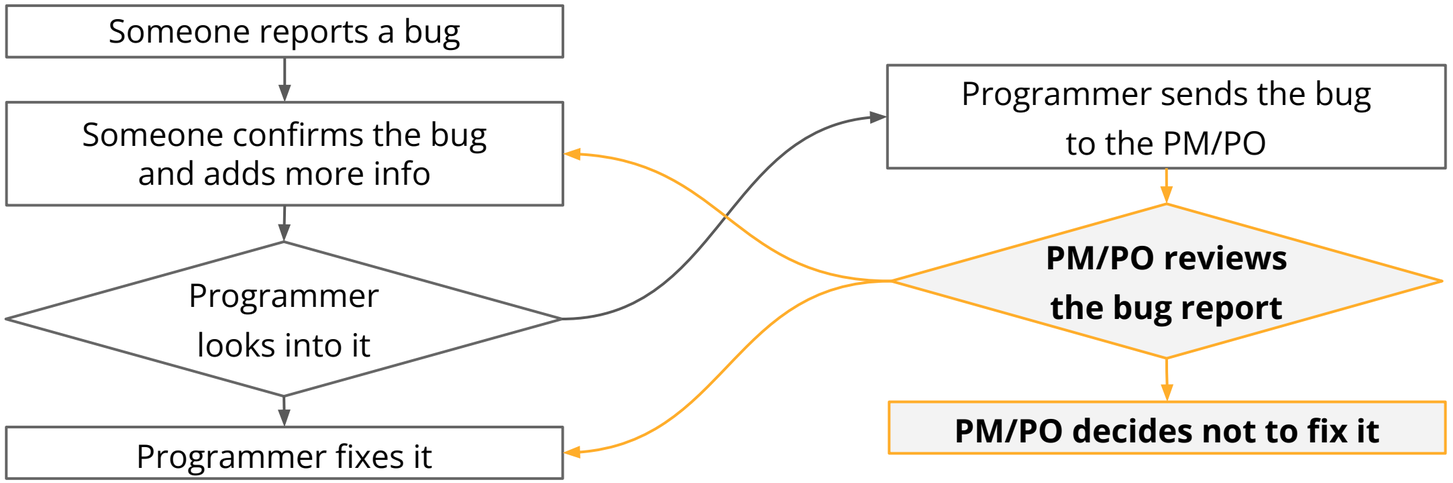


A programmer looks into it and

- fixes it,
- decides that a fix will take so long it needs management approval,
- recommends that it be deferred (fix it later), or
- determines (argues) it is not a bug.

The process of deciding which bugs to fix and which to leave in the product is called *bug triage*.

Summary of a Bug Workflow

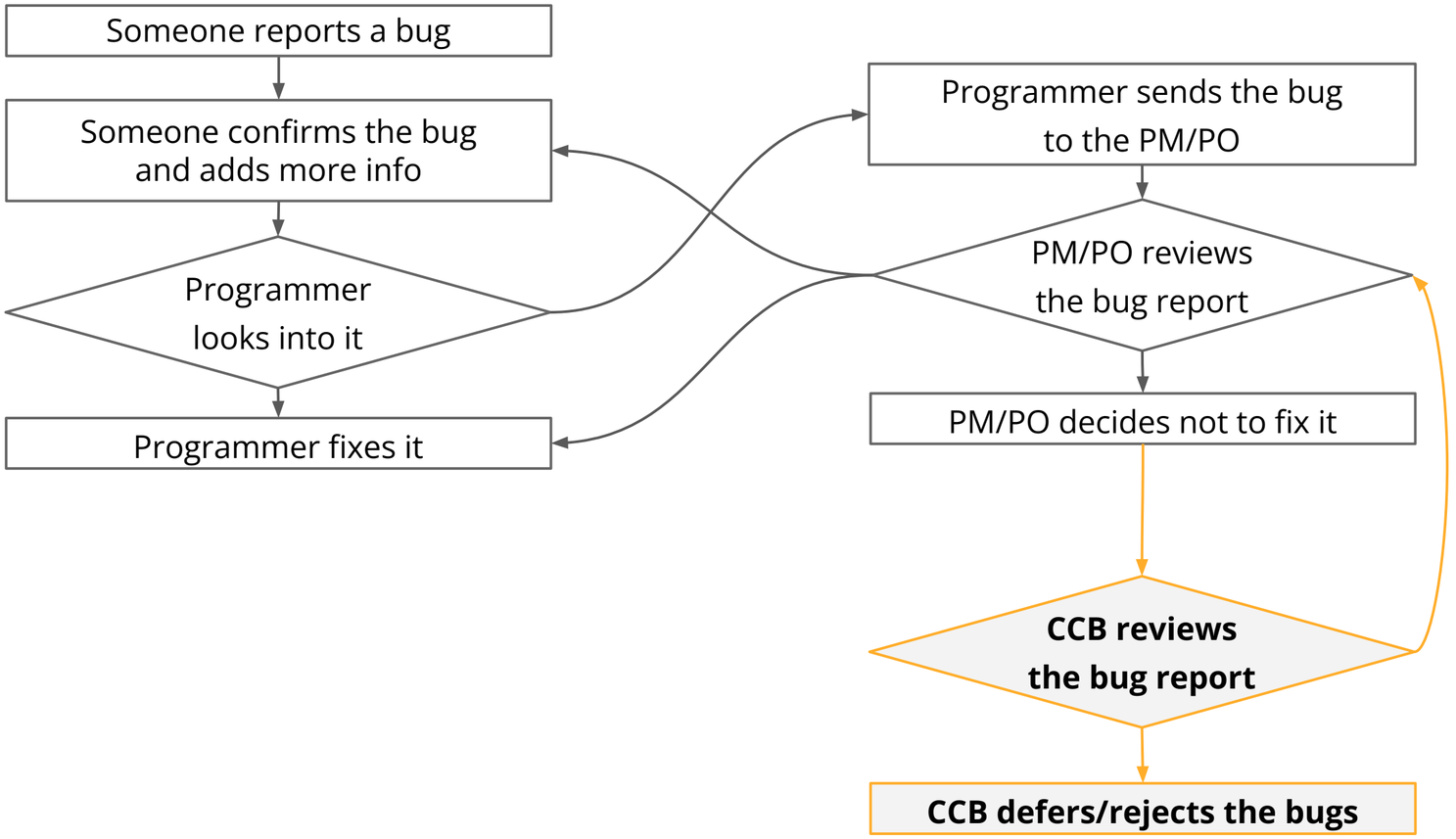


Project manager/product owner (PM/PO) may:

- prioritize the unfixed bugs,
- ask for more information or
- decides not to fix it either because “it’s a feature” or too expensive.

The Project Manager (PM) has the authority to decide what order the project’s tasks are done and which tasks are dropped if the project runs out of time. In Agile projects, the Product Owner (PO) usually prioritizes and/or drops backlog items and bugs based on direct feedback from the customer.

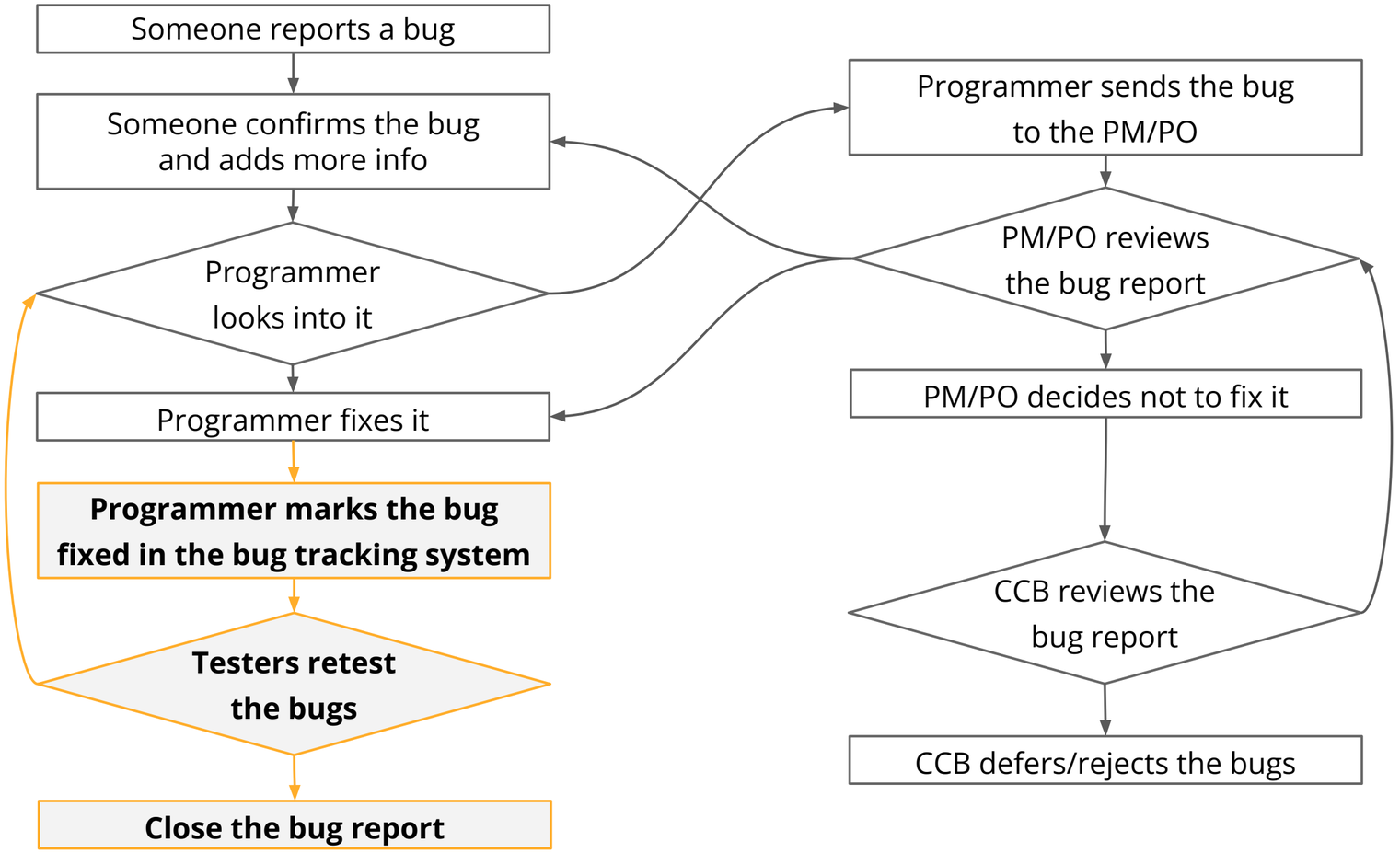
Summary of a Bug Workflow



- The project team (representatives of the key stakeholder groups) reviews bugs that seem expensive or that are deferred.
- This is often called the triage team or the Change Control Board (CCB)
- The CCB makes “final” decisions to defer or reject unfixed bugs.
- In Agile projects, the CCB might be replaced by the PO or by the customer themselves, who can make final decisions about bugs based on demo sessions or triage meetings with the whole team.

Few (or no) companies fix every bug they find. What separates great companies from irresponsible ones is the level of care and insight in their triage process.

Summary of a Bug Workflow

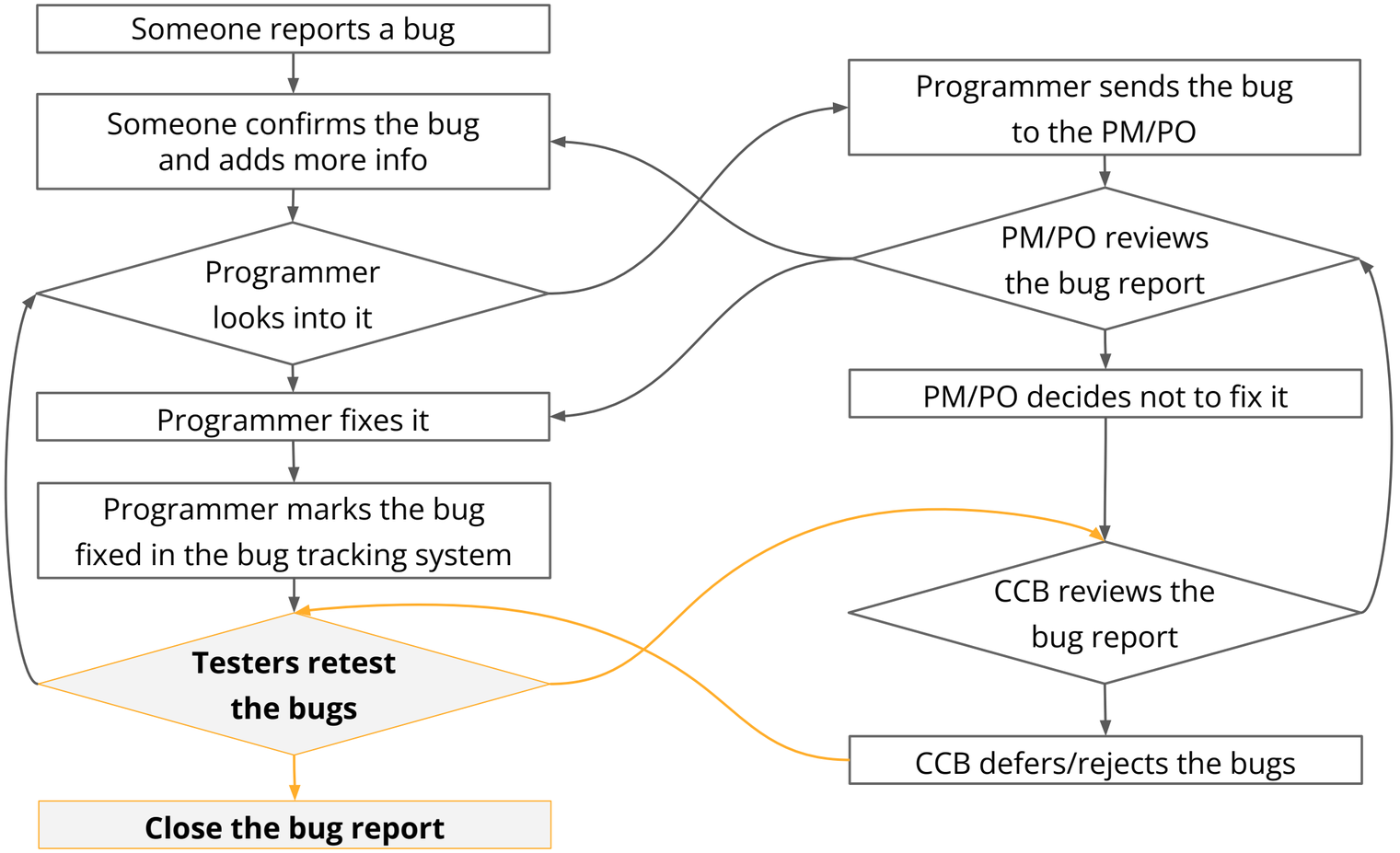


The test group retests the fixed bugs:

- Did the programmer actually fix the bug?
  - Just this particular bug or similar bugs in other parts of the code?
  - Just the reported symptoms or the full underlying problem?
- Did the fix cause other side effects?

Whenever you check a bug fix, vary your tests to explore the scope of the fix.

Summary of a Bug Workflow



Retest bugs marked as deferred, irreproducible or rejected and

- agree to close them, or
- appeal the no-fix decision: add new information to the bug report and ask the project manager and the triage team to fix the bug

Deciding which bug deferrals/rejections to appeal is a critical task that impacts the credibility of the test group.

Bug Reports Are a Tester's Primary Work Product

A bug report is a tool you use to sell the programmer on the idea of spending her time and energy to fix a bug.

Bug Advocacy?

Bug reports are what people outside of the testing group will most notice and most remember of your work.

The quality of your communication drives the success of your reports.

It is very important to find a link between the problem you see and the objectives and concerns of the stakeholders who will base decisions on your reports.

"The best tester isn't the one who finds the most bugs or embarrassed the most programmers. The best tester is the one who gets the most bugs fixed."

Kaner, C., Falk, J., & Nguyen, H.Q. (2nd Edition, 1999). *Testing Computer Software.*

Common Answers (Software Error)

BBST ®
BUG ADVOCACY

- Doesn’t match specifications or written requirements
- Doesn’t match documentation
- Coding error (doesn’t do what the programmer intended)
- Doesn’t meet design objectives
- Doesn’t meet company standards
- Doesn’t meet industry standards
- Would embarrass the company
- Makes the product less salable
- Interferes with development, testability or revision of the product
- Interacts badly with other programs or components
- Generates incorrect results
- Generates confusing results
- Wastes the time of a user
- Any failure (misbehavior)
- Underlying error that causes a failure
- Anything that, if reported, would lead to a code change
- Failure to meet reasonable expectations of a user (Myers)
- Failure to meet reasonable expectations of a stakeholder
- A bug is something that bugs somebody (Bach)

Anything that causes an unnecessary or unreasonable reduction of the quality of a software product.

Consider an Example

Here's a defective program

```
1   INPUT A
2   INPUT B
3   PRINT A/B
```

The user enters 5 (for A) and 0 (for B) and the program says:

```
ARGH! ?DIVby0! I have a headache!
```

and then halts.

What is the

- **bug?**
- **failure?**
- **fault?**
- **error?**
- **critical condition?**
- **defect?**

What's the "Standard" Answer?

IEEE Standard 610.12-1990 defines **ERROR** as:

"(1) The difference between a computed observed or measured value (1) The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result.

(2) An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program.

(3) An incorrect result. For example, a computed result of 12 when the correct result is 10.

(4) A human action that produces an incorrect result. For example, an incorrect action on the part of a programmer or operator.

Note: While all four definitions are commonly used, one distinction assigns definition 1 to the word 'error,' definition 2 to the word fault,' definition 3 to the word 'failure,' and definition 4 to the word 'mistake.' "

What's the "Standard" Answer?

IEEE Standard 610.12-1990 defines **FAULT** as:
"An incorrect step, process, or data definition in a computer program. Note: This definition is used primarily by the fault tolerance discipline. In common usage, the terms 'error' and 'bug' are used to express this meaning."

I've also seen "fault" used to refer to the behavioral failure.

IEEE Standard 610.12-1990 defines **FAILURE** as:
"The inability of a system or component to perform its required functions within specified performance requirements. Note: The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error)."

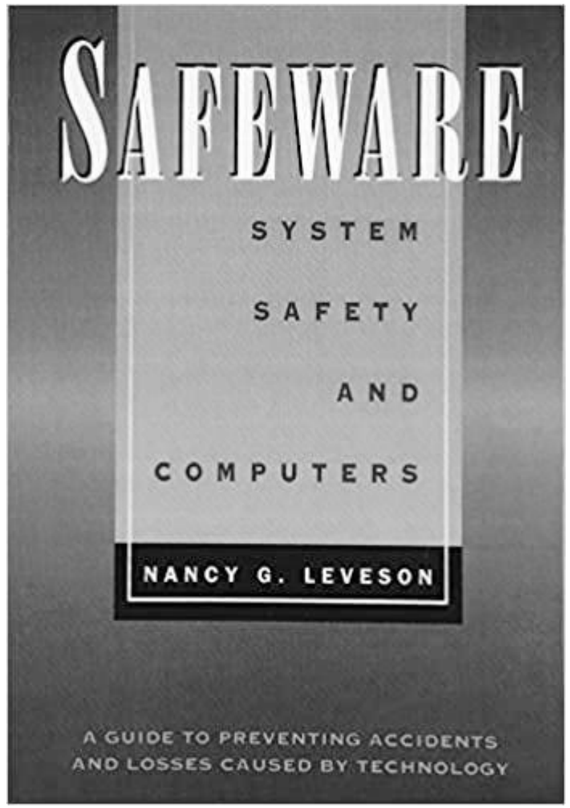
The Usage in THIS Course

- **An error (or fault) is something wrong with the product, such as a design flaw, an incorrect internal data value, or a coding error.**
- **The failure is the program's actual incorrect or missing behavior.**

The Usage in THIS Course

BBST ®
BUG ADVOCACY

- An **error**
- won't yield a **failure**
- without the **conditions** that trigger it.
    - Example, if the program yields 2+2=5 only the 10th time you use it, you won't see the error before or after the 10th use.
- In a test that yields a failure, a **critical condition** is a data value or step that is essential for revealing the problem)



Nancy Leveson (1995) draws useful distinctions between errors, hazards, conditions, and failures in *Safeware.*

The Definitions in This Course

A symptom is a behavior that suggests an underlying problem.

- For example the program responds a little example, the program responds a little more slowly than usual or a small part of the screen briefly flickers.
- Minor symptoms sometimes get worse (steadily deteriorating performance). Minor symptoms can be important clues for troubleshooting hard-to-reproduce bugs.
- Sudden and catastrophic bridge collapses and building failures usually are preceded by many innocuous "cosmetic" cracks. You can't know how bad a bug is just from a symptom.

The Definitions in This Course

The word *defect* might refer to the failure or to the underlying error.

However, because this term has significant legal implications, it should be used rarely, with great care, and preferably, in accordance with the local laws and company standards that were reviewed by counsel.

Many lawsuits rest on the question, "Can we prove this product was defective?"

The Definitions in This Course

BBST ®
BUG ADVOCACY

Software Error (or Bug) is anything that causes an unnecessary or unreasonable reduction of the quality of a software product.

Not every limitation on quality is a bug.

The Definitions in This Course

Not every limitation on quality is a bug.

What Is Quality?
Leading Definitions

- Conformance with requirements (Philip Crosby)
    - Actual requirements, which may or may not be what's written down.
- The totality of features and characteristics of a product that bear on its ability to satisfy a given need (American Society for Quality)
- The total composite product and service characteristics of marketing, engineering, manufacturing and maintenance through which the product and service in use will meet expectations of the customer (Armand V. Feigenbaum, Total Quality Control, Fortieth Anniversary Edition)

Note the absence of "conforms to specifications."

See Ishikawa. (1985). *What Is Total Quality Control?: The Japanese Way.*

What Is Quality?

BBST ®
BUG ADVOCACY

“Quality is fitness for use.” (Joseph Juran)

Joseph Juran distinguished between Customer Satisfiers and Dissatisfiers as key dimensions of quality:

Customer Satisfiers

- the right features
- adequate instruction

Dissatisfiers

- unreliable
- hard to use
- too slow
- incompatible with customer’s equipment

Which is the more important for quality?

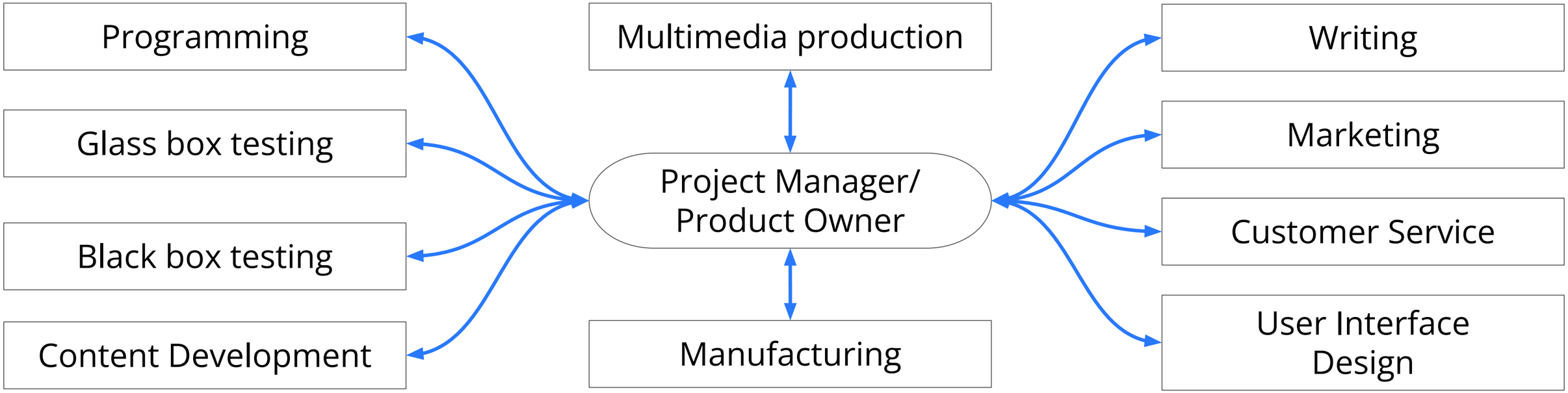
1. **happy customers?**
2. **low rate of problems?**
3. **adherence to a quality management process?**

Quality Is Multidimensional

All of these attributes tie to value:

- reliability
- usability
- maintainability
- testability
- salability
- functionality/capability
- speed of operation
- scalability
- localizability
- documentability
- trainability
- technical-supportability

Quality—According to WHO?



When you sit in a project team meeting, discussing a bug, a new feature, or some other issue in the project, you must understand that each person in the room has a different vision of what a “quality” product would be. Fixing bugs is just one issue.

Different People, Different Quality

BBST ®
BUG ADVOCACY

Localization Manager	A good product is easy to modify for another country, language and culture. Few experienced localization managers would consider acceptable a product that must be recompiled or relinked to be localized.
Tech Writers	A good product is easy to explain. Anything confusing, unnecessarily inconsistent, or hard to describe has poor quality.
Marketing	Good products drive people to buy them and encourage their friends to buy them. Adding desirable new features improves quality.
Customer Service	Good products are supportable: designed to help people solve their own problems or to get help quickly.
Programmers	Good code is maintainable, easy to understand, fast and compact.
Testers:	??? What do you think ???

If two people differ in how they perceive what is valuable or expensive, they differ in what drives their perception of quality from high to low.

So, What IS Quality?

BBST ®
BUG ADVOCACY

I like Gerald Weinberg's definition:

"Quality is value to some person."

But consider the implications:

- Anything that reduces the value reduces the quality.
- Quality is subjective. What's valuable for you is maybe not so valuable for me.
- A bug report can describe a problem that is perceived as serious by one person and trivial by the other—and both can be right.
- The essence of a good description of the severity of the problem is the mapping to reduction of stakeholder value.

The Definitions in This Course

Software Error (or Bug) is an attribute of a software product

- **that reduces its value to a favored stakeholder**
- **or increases its value to a disfavored stakeholder**
- **without a sufficiently large countervailing benefit.**

(Is this equivalent to “Anything that causes an unnecessary or unreasonable reduction of the quality of a software product”?)

“A bug is anything about the product that threatens its value.”

James Bach

Michael Bolton

Summing Up

BBST ®
BUG ADVOCACY

1. A bug report is an assertion that a product could be better than it is.
2. The idea of “better” is subjective—what’s better for one stakeholder might be worse for another.
3. One reason for the variation is the multidimensionality of every product. A feature, or a more general attribute of a product, might be more important to one person than to another. Bugs that weaken the feature (or attribute) are those more costly to the first person than the other.
4. A bug report is **justified** if it exposes a problem that does in fact reduce value for a stakeholder with influence.

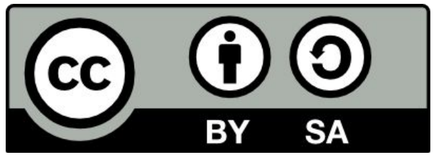
Black Box Software Bug Advocacy
Lecture 2
Effective Advocacy: Making People Want to Fix the Bug

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Rebecca L. Fiedler, M.B.A., PH.D.

Retired, President of Kaner, Fiedler & Associates



Course Overview: Fundamental Topics

BBST ®
BUG ADVOCACY

1. Basic Concepts
2. Effective Advocacy: Making People Want to Fix the Bug 
3. Writing Clear Bug Reports
4. Anticipating and Dealing With Objections: Irreproducible Bugs
5. Bugs that Could Be Dismissed as Unreasonable, Unrealistic, or Unimportant
6. Credibility and Influence

It’s Not Only About Reporting the Bug

- Client experienced a wave of serious product recalls (defective firmware)
  - Why were these serious bugs not found in testing?
    - **They WERE found in testing AND reported**
  - Why didn’t the programmers fix them?
    - **They didn’t understand what they were reading**
  - What was wrong with the bug reports?
    - **The problem is that the testers focused on creating reproducible failures, rather than on the quality of their communication.**
- Looking over 5 years of bug reports, I could predict fixes better by clarity/style/attitude of report than from severity.

But There Are Tradeoffs

1. The more time you spend on each bug, the fewer bugs you have time to find and report.
2. If you spend lots of troubleshooting time getting more information for the programmers, how much time does this save them?
    - *If an hour of your investigative time saves the programmer 10 minutes, is this a cost-effective allocation of resources?*

At some companies, testers report bugs as quickly as they can, providing extra troubleshooting only when the programmers dismiss the bug or can't find it.

Bug Advocacy = Selling Bugs

Time is in short supply. People are overcommitted.

If you want someone to fix your bug, you have to make them want to do it.

- *Your* bug? (Someone else made the bug, but once you find it, it's yours too.)

The art of motivating someone to do something that you want them to do is called *sales*.

Your task is to communicate effectively with human decision-makers. Purely technical cost/benefit tradeoffs miss the bigger picture.

Bug Advocacy

It’s not just about reporting bugs.

- It’s about presenting a bug in its strongest (honestly described) light.
- It’s about presenting a bug in a way that connects with the concerns of stakeholders with influence—and if one particular stakeholder will be most affected, by making sure she gets the message clearly.
- It’s about communicating so well that your report enables good decision-making.

The best tester isn’t the one who finds the most bugs or embarrasses the most programmers.
The best tester is the one who gets ~~the most~~ THE RIGHT bugs fixed.

How To Sell Bugs

Sales revolves around two fundamental objectives:

- **Motivate the buyer**
  - Make her WANT to fix the bug.
- **Overcome objections**
  - Get past her reasons and excuses for not fixing the bug.

Today we focus on writing a motivating report. Later, we consider the objections.

Motivating the Bug Fixer

Some programmers want to fix a bug if:

- It looks really bad.
- It will affect lots of people.
- Getting to it is trivially easy.
- It is a breach of contract.
- A bug like it has embarrassed the company, or a competitor.
- It looks like an interesting puzzle and piques the programmer's curiosity.
- Management (that is, someone with influence) has said that they really want it fixed.
- You said you want this particular bug fixed, and the programmer likes you, trusts your judgment, is susceptible to flattery from you, or owes you a favor.

Researching the Failure Conditions

When you run a test and find a failure, you're looking at a **symptom**, not at the underlying error.

You may or may not have found the best example of a failure that can be caused by the underlying fault.

Therefore you should do some follow-up work to try to prove that a coding error:

- is more serious than it first appears.
- is more general than it first appears.

Refresher on Terminology

- **Error**: the mistake in the code
- **Failure**: the misbehavior of the program
- **Critical conditions**: the data values, environmental conditions, and program steps that are essential for eliciting the failure
- **Symptom**: a minor misbehavior that warns of an error that can cause a much more serious failure

More Serious Than It First Appears

Look for follow-up errors.

When a program fails because of a coding error:

- The program is in a state that the programmer did not intend and probably did not expect.
- There might also be data with “impossible” or unexpected values.

The program is now in a vulnerable state.

- Even more vulnerable because error-handling code is often buggy.

Keep testing. The real impact of the underlying fault might be much worse, such as system crash or corrupted data.

At the start of follow-up testing, consider whether you need to conserve your state (and how to do it).

Follow-Up Testing for Severity

I do four types of follow-up testing:

1. Vary my behavior
    - change what I do as I test
2. Vary the options and settings of the program
    - change settings of the application under test
3. Vary data that I load into the program
    - different startup files or other data not directly involved in the test
4. Vary the software and hardware environment
    - e.g. operating system, peripherals, external software that interacts with this application

Follow-Up 1: Vary Your Behavior

Example: A program unexpectedly but slightly scrolls the display when you add two numbers:

- The task is entering numbers and adding.
- The failure is the scrolling.

Follow-Up 1: Vary Your Behavior

Try simple repetition.

- Bring it to the failure case again (and again). If the program fails when you do X, then do X many times. Is there a cumulative impact?

Look at timing.

- Enter the numbers more quickly or change speed of your activity in some other way.

Follow-Up 1: Vary Your Behavior

BBST ®
BUG ADVOCACY

Try things related to the task that failed.

- Try tests that:
  - affect adding or
  - that affect the numbers
  - maybe try negative numbers, for example.
- Try the program's multiply feature (it probably uses much of the same code as the add feature).

Follow-Up 1: Vary Your Behavior

Try combinations.

- If the failure is, “Do X, see the scroll”
  - Do Y then do X
  - Do X, then Z, then X
  - (If the scrolling gets worse or better in one of these tests, follow that up, you’re getting useful information for debugging.)

Combinations might involve features (Y, Z) that are:

- Related to the feature under test (adding and display)
- Or to the failure (scrolling)

Follow-Up 1: Vary Your Behavior

Try things related to the failure (scrolling).

- Do huge numbers or sums change the amount of scrolling?
- Try scrolling first, then adding. Try repainting the screen (scrolling is a screen-thing), then adding. Try resizing the display of the numbers, then adding.

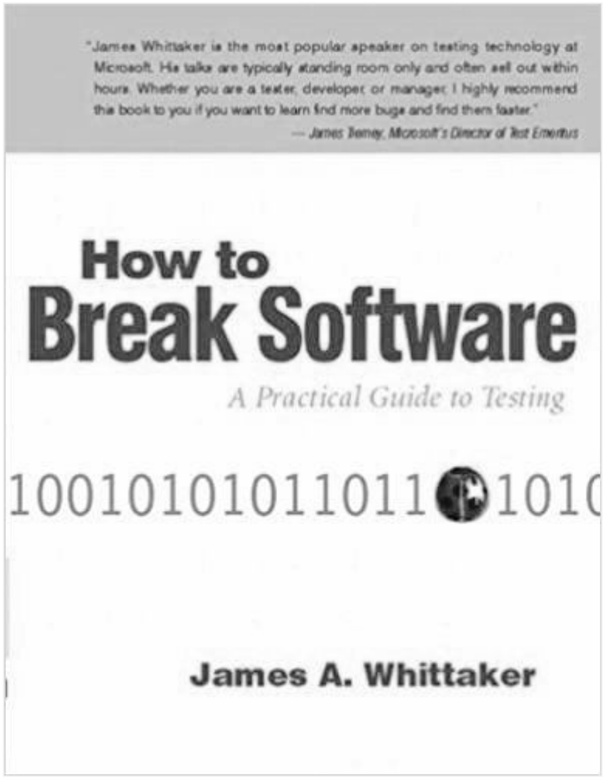
Follow-Up 1: Vary Your Behavior

BBST BUG ADVOCACY

And try the usual exploratory attacks (quicktests).

Continuing the example of unexpected scrolling after adding:

- Can we add more than two numbers? What's the maximum?
- Try the biggest numbers we can add
- Try some interference tests. Stop or pause the program or swap it just as the program is failing.
- Or try it while the program is doing a background save. Does that cause data loss corruption along with this failure?



See Whittaker, J., (2002), *How to Break Software: A Practical Guide to Testing*

Follow-Up 2: Vary Options & Settings

BBST ®
BUG ADVOCACY

Change the state of the program under test.

- does that change the failure?

In this case, keep the steps that demonstrate the failure the same.

Change the program, not what you do with the program.

For example,

- Change the values of persistent variables (any program options that seem potentially relevant)
- Change how the program uses memory (if that's a **program** setting)

Follow-Up 3: Vary Data Files

Change data that I load into the program.

- different startup files
- or other data not directly involved in the test

(If I'm varying the data directly used by the program, that's probably part of Follow-Up 1: Changing what I do)

Follow-Up 4: Vary the Configuration

BBST ®
BUG ADVOCACY

You might trigger a more serious failure if you replicate with less memory, a different printer, more device interrupts, etc.

- If it might involve timing, use a really slow (or fast) computer or peripheral.
- In our scrolling example, consider different settings on a wheel mouse that does semi-automated scrolling.
- If there is a video problem, try other resolutions on the video card. Try displaying MUCH more (less) complex images. If a recent OS update changed video handling, try the old OS version.

Follow-Up 4: Vary the Configuration

Note that we are not:

- checking standard configurations
- trying to understand how broad the range of circumstances is that produces the bug.

What we're asking is whether there is a particular configuration that will demonstrate this bug more spectacularly.

Researching the Failure Conditions

When you run a test and find a failure, you're looking at a **symptom**, not at the underlying error.

You may or may not have found the best example of a failure that can be caused by the underlying fault.

Therefore you should do some follow-up work to try to prove that a defect:

- is more serious than it first appears.
- **is more general than it first appears.**

"Is it more general?" means, "Will more people see it, more often, or under a wider range of circumstances, than the first failure we saw suggests?"

Researching the Failure Conditions

Dealing with extreme-value tests:

- We test at extreme values because these are the most likely places to show a defect.
- Once we find the defect, we don't have to stick with extreme value tests.
- We can look for the same failure under more general conditions.

"Is it more general?" means, "Will more people see it, more often, or under a wider range of circumstances, than the first failure we saw suggests?"

Showing a Bug Is More General

BBST ®
BUG ADVOCACY

Uncorner your corner cases.

- Corner case—a perceived-to-be ridiculously extreme test

To uncorner an extreme value test, try mainstream values.

- These are values (e.g. mid-range) that the program should handle easily.
- If you replicate the bug, write it up with the mainstream settings. This will be a very credible bug report.

A true "corner case" is a test that uses extreme values from at least two variables at the same time. (This may be a perfectly plausible test.)

Corner Cases

If mainstream values don't yield failure, troubleshoot around the extremes.

- Is the bug tied to this one extreme case?
- Is there a small range of cases?
- In your report, identify the narrow range that yields failures.
- A narrow-range bug might be deferred. That might be the right decision. Other bugs might be more critical.

The best tester isn't the one who finds the most bugs or embarrasses the most programmers.
The best tester is the one who gets ~~the most~~ THE RIGHT bugs fixed.

If You DO Have to Defend a Corner Case

- Sometimes, it's too hard to retest a corner case.
    - Example: customer-reported failure on a configuration you don't have in the lab (common with mainframes)
- Sometimes, you can only trigger the failure with an extreme case, but you think there is a more general problem, you just don't know how to prove it.
- Ask the programmer who recommends deferral:
    - Do you know the CAUSE of this?
    - Do you know the deferral is safe?
- The fact is, they have better diagnostics. Sometimes, the project team has to rely on them, not you.

Researching the Failure Conditions

When you run a test and find a failure, you're looking at a **symptom**, not at the underlying error.

You may or may not have found the best example of a failure that can be caused by the underlying fault.

Therefore you should do some follow-up work to try to prove that a defect:

- is more serious than it first appears.
- **is more general than it first appears.**

Another more kind of generality:

"How many systems does this run on?"

Look for Configuration Dependence

Question: How many programmers does it take to change a light bulb?

Answer: What's the problem? The bulb at my desk works fine!

Bugs that don't fail on the programmer's machine are much less credible (to that programmer).

If a bug is configuration dependent, set the programmer's expectations:

- Say that it only happens on some systems.
- Describe the relevant configuration variables that you know about.

When a programmer tests a bug she knows is configuration-dependent, if it doesn't fail on her machine, she'll know that she should try it on a different one.

Configuration Dependence

Does this failure happen on

- all systems that can **run this program?**
- all systems that run this program **on this operating system?**
- all systems that run this program **on this operating system with this much memory, that kind of display, this version of the video driver, a firewire drive connected, that firewall, and this virus checker running?**

A failure that shows up on most systems will be seen as more important than one that will show up on only very few.

Many errors, such as design errors, are obviously configuration-independent and so there is no point running configuration tests on them.

Testing for Configuration Dependence

It's common to test on 2 machines, Machines 1 and 2.

- Machine 1 is your powerhouse: latest processor, updated operating system, fancy printer, video card, USB devices, huge hard disk, lots of RAM, fast broadband, etc.
- Machine 2 is the turtle: slower processor, different keyboard, different video driver, barely enough RAM, slow, small hard drive, slower network connection.
- Many people do most testing on the turtle using the powerhouse for replication and test design.
- When you find a defect on Machine 2, replicate on Machine 1 (preserving state on Machine 2).
    - If you get the same failure, write it up.
    - If you don't get the same failure, you have a configuration-dependent bug. Now do troubleshooting.

Bug Reporting, With 2 Machines

When you test with two machines, you can check your report while you write it.

- Write the steps, one by one, on the bug report form at Machine 1.
- As you write them, try them on Machine 2.
- The final report has been tested. If you follow your steps literally, then someone else following your steps will probably get the results (see the bug) you are trying to describe.

It is good general practice to replicate every coding error on a second system.

Researching the Failure Conditions

When you run a test and find a failure, you're looking at a **symptom**, not at the underlying error.

You may or may not have found the best example of a failure that can be caused by the underlying fault.

Therefore you should do some follow-up work to try to prove that a defect:

- is more serious than it first appears.
- **is more general than it first appears.**

One more kind of generality:

"How new is it?"

Follow-Up: Bug New to This Version?

BBST ®
BUG ADVOCACY

Has this bug been in the program through several releases?

- Check its user-complaint history
- If no one complained, it probably won't be fixed

Is this bug new?

- Especially important in maintenance updates. Bugs won't be fixed unless they were (a) scheduled to be fixed because they are critical or (b) side effects of the code changes.
- An old bug should be treated as new if it behaves differently or shows up under new conditions.

Adding Information Beyond Test Results

- Comparisons with competitors
- Predicted criticisms from the press
- Usability weaknesses
- Lost value because it's too hard to achieve a benefit that programmers don't think is so important
- Predicted support costs
- Other implications for sales, support, legal, etc.

Who is your credible source?

- The person **your team believes** is the expert is the ideal person to quote (or have speak).
- Provide data from credible sources.

No one expects the tester to be an expert in marketing or human factors - even if you are an expert - and so anything you say might be dismissed as uninformed opinion.

Summing Up

1. High impact reporting requires follow-up testing to present the bug in its strongest light:
    - Look for the most serious failure
    - Check the range of conditions under which it can be replicated
    - Especially for maintenance updates, check whether this bug is a new side-effect of recent code changes.
2. To further motivate people to fix bugs:
    - Look for market data (impact in the field?). **Use credible sources.**
    - Manage your relationships with other stakeholders with care.

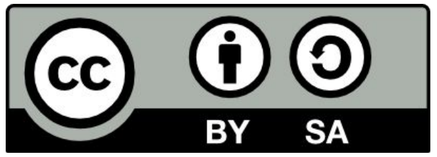
Black Box Software Bug Advocacy
Lecture 3
Writing Clear Bug Reports

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Rebecca L. Fiedler, M.B.A., PH.D.

Retired, President of Kaner, Fiedler & Associates



Course Overview: Fundamental Topics

BBST ®
BUG ADVOCACY

1. Basic Concepts
2. Effective Advocacy: Making People Want to Fix the Bug
3. Writing Clear Bug Reports 
4. Anticipating and Dealing With Objections: Irreproducible Bugs
5. Bugs that Could Be Dismissed as Unreasonable, Unrealistic, or Unimportant
6. Credibility and Influence

The Bug Tracking System

Testers report bugs into a bug tracking system.

- The system (implicit culture or explicit policies, procedures, mission) determine such things as:
  - what types of problems they submit
  - what details go into the reports
  - who has access to the data, for what purpose
  - what summary reports and statistics are available

Mission of Bug Tracking Systems

Having a clear mission is important.

Without one, the system will be called on to do

- Too many things (making bug reporting inefficient)
- Annoying things
- Contradictory things

Mission of Bug Tracking Systems

Given a primary mission, all other uses of the system are secondary and must be changed or modified if they interfere with the primary mission.

Examples of issues to consider:

- Auditing
- Tracing to other documents
- Personal performance metrics (reward or punishment)
- Progress reporting
- Schedule reality checking
- Scheduling (when can we ship?)
- Archival bug pattern analysis

Anything that does not directly flow from the mission is a side issue. Anything that makes achieving the mission harder is counterproductive.

Mission of Bug Tracking Systems

The mission that I prefer is this:

A bug tracking process exists for the purpose of getting the right bugs fixed.

Mission of Bug Tracking Systems

Given a primary mission, all other uses of the system are secondary and must be changed or modified if they interfere with the primary mission.

Examples of issues to consider:

- Auditing
- Tracing to other documents
- **Personal performance metrics (reward or punishment)**
- Progress reporting
- Schedule reality checking
- Scheduling (when can we ship?)
- Archival bug pattern analysis

Anything that does not directly flow from the mission is a side issue. Anything that makes achieving the mission harder is counterproductive.

See Austin, R., (2013), *Measuring and Managing Performance in Organizations*.

Getting the Right Bugs Fixed

BBST ®
BUG ADVOCACY

- Capture all of the right reports
- Capture all the relevant details
- Encourage clarity and collegiality, cooperation and rational evaluation, in writing and responding to bugs
- Support dialog (technical discussion, evaluative discussion)
- Allow multiple viewpoints, multiple evaluations
- Accessible to every stakeholder with influence
- Nothing gets swept under the carpet
- Nothing gets accidentally lost

From here, I assume the mission is getting the right bugs fixed.

To Report a Bug Well: Replicate

You must either:

- make sure the failure is reproducible by anyone who follows the steps in your report or
- tell the reader that the problem is configuration-dependent or intermittent and provide relevant details.

- **Replicate** ←
- **Isolate**
- **Maximize**
- **Generalize**
- **Externalize**
- **Neutral tone**

To Report a Bug Well: Isolate

- You have a list of the steps you took to show the error. You're now trying to shorten the list
- What are the critical conditions?
- Write a report that includes the minimum set of steps needed to replicate the failure
- Include all the steps needed
- Keep it simple: only one failure per report
- If a sample test file is essential to reproducing a problem, reference it and attach the test file

- **Replicate**
- **Isolate** ←
- **Maximize**
- **Generalize**
- **Externalize**
- **Neutral tone**

Isolate the Failure:

Eliminate Unnecessary Steps (1)

- Try taking out individual steps or small groups of steps and see whether you still replicate the failure.
- Sometimes it’s not immediately obvious what can be dropped from a long sequence of steps in a bug.

The best report is the one that reaches the failure in the shortest, simplest set of steps.

Isolate the Failure:

Eliminate Unnecessary Steps (2)

- Look carefully for any hint of an error as you take each step. Examples:
  - Error messages (you got a message 10 minutes ago. The program didn't fully recover from the error - the problem you see now is caused by that poor recovery.)
  - Display oddities, such as a flash, repainted screen, cursor that jumps back and forth, multiple cursors, misaligned text, slightly distorted graphics, doubled characters, omitted characters, or display droppings (pixels that are still colored even though the graphic that contained them was erased or moved).

Look for symptoms that provide early warning of the more dramatic failure to follow. The steps that trigger the symptoms are usually critical.

Isolate the Failure:

Eliminate Unnecessary Steps (3)

- Look carefully for any hint of an error as you take each step:
  - Delays or unexpectedly fast responses
  - Noticeable change in memory used
  - Sometimes the first indicator the system is working differently is that it sounds a little different than normal
  - An in-use light or other indicator that a device is in use goes unexpectedly on or off
  - Debug messages—turn on your system's debug monitor (if it has one)—see if/when a message is sent to it

Isolate the Failure:

Eliminate Unnecessary Steps (4)

Once you find what looks like a critical step, try to eliminate almost everything else from the bug report.

- Go directly from that step to the one(s) that appear to be the final trigger(s) for the failure.

If this approach doesn't work, try taking out individual steps or small groups of steps more arbitrarily and see whether you still replicate the failure.

Isolate the Failure:

Two Failures ➔ Two Reports

Reports with two problems:

- Description is longer and harder to follow, and therefore
  - less likely to be addressed and
  - more likely to be misunderstood
- Summary line is often vague (says “fails” instead of describing the failure)
- Half the report gets fixed and it gets closed

When you report related problems on separate reports, it is a courtesy to cross-reference them.

If half the bug could be fixed and the other half not, report two bugs.

To Report a Bug Well: Maximize

Follow-up testing to see if you can demonstrate a worse failure than the initial symptom:

- Vary your actions
- Vary the program's settings
- Vary the stored data you give the program
- Vary the environment

- **Replicate**
- **Isolate**
- **Maximize** ←
- **Generalize**
- **Externalize**
- **Neutral tone**

To Report a Bug Well: Generalize

BBST ®
BUG ADVOCACY

- Uncorner your corner cases (show it fails (or doesn't) under less extreme conditions).
- Show it fails (or doesn't) on a broad range of systems.

- **Replicate**
- **Isolate**
- **Maximize**
- **Generalize** ←
- **Externalize**
- **Neutral tone**

To Report a Bug Well: Externalize

- Switch focus from the program to the stakeholders
  - What are the consequences of this failure?
  - Is comparative data available?
    - Historical support data for similar bugs?
    - Other historical cost data?
    - Competitors' problems?
  - Have people written about problems like these in this or other products?
  - What benefits does this failure interfere with?
  - Who would care about this failure and why?
    - Get them to help you understand what this costs them.

- **Replicate**
- **Isolate**
- **Maximize**
- **Generalize**
- **Externalize** ←
- **Neutral tone**

To Report a Bug Well: Neutral Tone

- Make the report easy to understand.
- Keep your tone neutral and nonantagonistic.
- Angry, blaming reports discredit the reporter.

- **Replicate**
- **Isolate**
- **Maximize**
- **Generalize**
- **Externalize**
- **Neutral tone ←**

Typical Fields in a Problem Report

Problem report number - must be unique	**Reproducible -** yes/no/sometimes/unknown	**Status -** tester fills this in: open/closed
Date reported - date of initial report	**Release number -** like Release 2.0	**Resolution -** fixed, deferred, etc.
Problem summary (problem title) - one-line summary of the problem	**Severity -** assigned by tester, some variation on small/medium/large	**Resolution version -** build identifier
Program (or component) name - the visible item under test	**Priority -** assigned by programmer/project manager/product owner	**Resolved by -** programmer, tester (if withdrawn by tester), etc.
Report type - e.g. coding error, design issue, documentation mismatch, suggestion, query	**Problem description and how to reproduce it -** step by step reproduction description	**Change history -** date stamped list of all changes to the record, including name and fields changed
Version (build) identifier - like version C or version 20000802a	**Suggested fix -** leave it blank unless you have something useful to say	**Key words -** use these for searching later, anyone can add to key words at any time
Configuration(s) - hardware and software configurations under which the bug was found and replicated	**Customer impact -** often left blank. When used, typically filled in by tech support or someone else predicting actual customer reaction (such as support cost or sales impact)	**Resolution tested by -** someone checks the bug and agrees it was fixed, or the non-fix resolution is defensible. Usually this is the original reporter, or a tester if originator was a non-tester
Reported by - original reporter's name. Some forms add an editor's name	**Assigned to -** shows who is currently working on the bug - it could be a developer working on a fix, a tester verifying a fix, etc.	**Comments -** free-form, arbitrarily long field, typically accepts comments from anyone on the project until the bug is resolved

Typical Fields: Problem Summary

One-line description of the problem, sometimes called the report title.

- Stakeholders will use it in when reviewing the list of bugs that haven't been fixed.
- Many stakeholders spend additional time only on bugs with "interesting" summaries.

The summary is the most important part of the report.

Typical Fields: Problem Summary

The ideal summary gives the reader enough information to help her decide whether to ask for more information. It should follow a FAILURE-WHEN structure:

- brief description that is specific enough that the reader can visualize the failure.
- brief indication of the limits or dependencies of the bug (how narrow or broad are the circumstances involved in this bug?)
- Some other indication of the severity (not a rating but helping the reader envision the consequences of the bug.)

Use a FAILURE – WHEN structure

Typical Fields: Report Type

These are common distinctions:

- **Coding Error:** The program doesn't do what the programmer expects it to do.
- **Design Issue:** It does what the programmer intended, but a reasonable customer might not like it.
- **Requirements Issue:** The program is well designed and implemented, but won't meet some stakeholder's needs.
- **Documentation/Code Mismatch:** Report this to the programmer (via a bug report) and to the writer (usually via a memo or a comment on the manuscript).
- **Specification/Code Mismatch:** Maybe the spec is right; maybe the code is right; sometimes they're both wrong.

This type of categorization avoids misunderstandings. If you're reporting design issues (for example), you don't want them misread as (erroneous) reports of coding errors.

Typical Fields: Report Type

- **Enhancement request:** A requirements issue--the program doesn't meet a need--but the person who labels it an enhancement request is saying that the stakeholders who set the budget should expect to pay extra if they want this.
- **Scope issue:** This project is governed by a contract with another organization or a negotiated agreement with management that lays out the tasks of the project. This enhancement request is "out of scope"—it requires an amendment to that contract.
- **Compatibility issue:** The misbehavior of the program is blamed on interaction with other software that this development team probably doesn't control.

These are sometimes recategorizations, assigned during triage, as ways of explaining why a bug won't be fixed or a schedule/cost target won't be met. Sometimes, they seem more like excuses than categorizations.

Typical Fields: Severity vs Priority

- Severity is the reporter's assessment of the badness of the bug.
- Some companies use two or three severity fields:
  - the tester's estimate
  - the project manager's estimate
  - the estimate of impact on the user (or other stakeholder impact) from a person who will most bear that cost (e.g. tech support impact)
- Priority is the project manager's decision about when the bug has to be fixed. Many factors, not just severity, drive this timing.

There is more information, and much more goodwill, if you let disagreeing staff each say their piece without having to overwrite the statement/rating of the other.

Typical Fields: Problem Description

BBST ®
BUG ADVOCACY

- First, describe the problem. What's the bug? Don't rely on the summary to do this -- some reports will print this field without the summary.
- Next, go through the steps that you use to recreate this bug.
  - Start from a known place (e.g. boot the program) and
  - Then describe each step until you hit the bug.
  - Take it one step at a time.
  - Number the steps.

The first list of steps should describe the shortest step-by-step path to the failure.

Typical Fields: Problem Description

Go through the steps that you use to recreate this bug.

1. Do this
2. Do this
3. Do this ➔ see that
4. Do this, etc.

You are giving people directions to a bug. Especially in long reports, people need landmarks. Tell them what they should see, every few steps or whenever anything interesting happens.

Typical Fields: Problem Description

- Describe the erroneous behavior.
  - If necessary, explain what should have happened.
    - Why do you think this is this a bug?
    - Describe what result you expected instead of the result you got.

Typical Fields: Problem Description

- List the environmental variables (configuration, etc.) that are not covered elsewhere in the bug tracking form.
- If you expect the reader to have any trouble reproducing the bug (special circumstances are required), be clear about them.

Typical Fields: Problem Description

Suppose the failure looks different under slightly different circumstances. Examples:

- The failure won't show up or is much less serious if you do step X between step Y and step Z
- The timing changes if you do two additional sub-tasks before hitting the final reproduction step
- The printer prints different garbage (instead of the garbage you describe) if you make the file a few bytes longer

Add a section that says "**Additional Conditions**" and describe, one by one, in this section the additional variations and the effect on the observed failure.

Typical Fields: Suggested Fix

Leave it blank unless you have something useful to say.

Typical Fields: Status and Resolution

Tester fills the status field in:

- **Open**
- **Closed**
- **Resolved**

The project manager owns the resolution field.

Common resolutions include:

- **Pending**: the bug is still being worked on.
- **Fixed**: the programmer says it's fixed. Now you should check it.
- **Cannot reproduce**: The programmer can't make the failure happen. You must add details, reset the resolution to Pending, and notify the programmer.
- **Deferred**: It's a bug, but we'll fix it later.

Typical Fields: Resolution

- **As Designed**: The program works as it should.
- **Need Info:** The programmer needs more info from you. She has probably asked a question in the comments.
- **Duplicate**: This is a repeat of another bug report (XREF it on this report)
    - I prefer to keep duplicates open until the duplicated bug closes.
    - Others consolidate duplicates and close all but the consolidator.
- **Withdrawn**: The tester who reported this bug is withdrawing the report.
- **INWTSTA**: I never want to see this again.

Typical Fields: Comments

- Free-form, arbitrarily long field, typically accepts comments from anyone on the project until the bug is resolved.
- Questions and answers about reproducibility, and closing comments (why a deferral is OK, or how it was fixed for example) go here.
    - Write carefully. Just like e-mail, it's easy to read a joke or a remark as a flame. Never flame.

Fields I Avoid: Unreliable Data

I have a rule:

> **If I can't manage the accuracy of a type of data, I don't want to record it.**

It's OK to create fields for other groups to fill out, if they want to fill them out.

- It's not OK for testers to create a field for other people to fill out, if those people:
  - won't fill them out, or
  - won't fill them out without nagging.
- Setting ourselves up to be nagging pests affects our general goodwill and credibility in the development group.

Garbage in

Garbage out

Fields I Avoid: Project Phase

Some groups record the development phase in which the bug was made and the phase when it was fixed.

- This assumes that development proceeds in phases (waterfall). It makes much less sense in a spiral or iterative model, when the staff can make a “requirements” error or “design” error after they’ve written most of the code.
- Even in a genuinely phased project, it is often difficult (especially for the test group) to determine when the mistake was made.

Garbage in

Garbage out

Fields I Avoid: Root Cause, Failure Module, Cost

The tester won't know where the error is. Is it bad data, a coding error, wrong algorithm choice, protocol mismatch with an external program?

The debugging programmers figure this out, but will they add the information to the bug report?

- Some testers attempt to reconstruct this info from source control records. This risks significant/frequent error.
- If the programmers won't gladly enter this data, don't collect it.

Similarly for cost of fixing the bug, do even the programmers know this? Will they voluntarily write it down?

Garbage in

Garbage out

Fields I Avoid: Cost Mitigated

How much future loss was mitigated by finding and fixing this bug?

- Sometimes, we can estimate this.
  - Some tech support managers can predict, for some bugs, that a given bug is going to cost $100,000 in tech support costs.
  - I've never met a manager who could do this accurately for most bugs.
- What should be reported if the support (or other) staff don't know or don't say what the cost will be?

Garbage in

Garbage out

Fields I Avoid: Employee Performance

- Who made the error? A report that names the bad person who made the bug is going to be taken personally by every programmer whose name is ever put on a bug report.
- How long did it take to fix the bug?
- How many bugs per week does this person report? or fix?

Let the project manager track his own staff performance in his own (private) notebook.

If you start counting these things, the people you are tracking will change their behavior in ways that improve their counts—but not necessarily ways that benefit the company.

If the mission of the system is getting the right bugs fixed, using the system for human performance reports is a direct conflict.

Summing Up

BBST ®
BUG ADVOCACY

1. RIMGEN gives you guidance about the types of research that you can do to improve your bug reports.
2. A good bug tracking system helps us get the right bugs fixed.

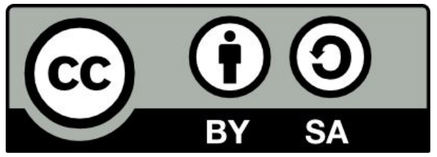
Black Box Software Bug Advocacy Lecture 4 - Anticipating and Dealing with Objections: Irreproducible Bugs

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Rebecca L. Fiedler, M.B.A., PH.D.

Retired, President of Kaner, Fiedler & Associates



Course Overview: Fundamental Topics

1. Basic Concepts
2. Effective Advocacy: Making People Want to Fix the Bug
3. Writing Clear Bug Reports
4. Anticipating and Dealing With Objections: Irreproducible Bugs
5. Bugs that Could Be Dismissed as Unreasonable, Unrealistic, or Unimportant
6. Credibility and Influence

Overcoming Objections

Programmers resist spending time on a bug if:

- **The programmer can't replicate the failure.** ←
- The programmer doesn't understand the report.
- It will take too much work to figure out what the reporter is complaining about and what steps he actually took.
- It seems unrealistic (e.g. "corner case" or requires a complex sequence of improbable steps)
- It's (allegedly) not a bug, it's a feature.
- It will take a lot of work to fix the bug or will introduce too much risk into the code.
- It is perceived as unimportant: No perceived customer impact, minor failure, unused feature.
- Management doesn't care about bugs like this.
- The programmer doesn't like/trust you (or the customer who is complaining about the bug).

Non-Reproducible Failures

Always report non-reproducible failures.

- Identify the problem as non-reproducible
- Describe the failure as precisely as you can
- If you see an error message, copy its words/numbers down exactly
- If the screen changes, note the details of the screen change.
- If you are recording output to other devices or systems, include all messages (and which things stayed silent)
- Describe the ways you tried to reproduce the failure

These can help the programmer identify specific points in the code that the failure did or did not pass through.

Always report non-reproducible failures, but report them carefully.

Can You Reproduce the Problem?

BBST ®
BUG ADVOCACY

Always provide this information even if it's not on the form.

- Never call a bug reproducible unless you've replicated it.
- In some cases, the appropriate status is "not attempted."
- If you can't recreate the bug, say so. Describe your attempts to recreate it.
- If the bug appears sporadically and you don't yet know why, say "Sometimes" and explain.
- You may not be able to try to replicate some bugs. Example: customer-reported bugs where the setup is too hard to recreate.

If the tester says a bug is reproducible and the programmer says it's not, the tester might have to recreate it in the presence of the programmer.

Non-Reproducible Failures

When you find a bug, preserve the machine's state. Try to replicate on a different machine. When you realize you can't reproduce a bug:

- Write down everything you can remember.
- Do it now, before you forget even more.
- As you write, ask yourself whether you're sure that you did this step (or saw this thing) exactly as you are describing it. If not, say so. Draw these distinctions right away. The longer you wait, the more you'll forget.
- Maybe the failure was a delayed reaction to something you did before starting this test or series of tests. Before you forget, note the tasks you did before running this test.
- Check the bug tracking system. Are there similar failures? Maybe you can find a pattern.
- Talk to the programmer and/or read the code.

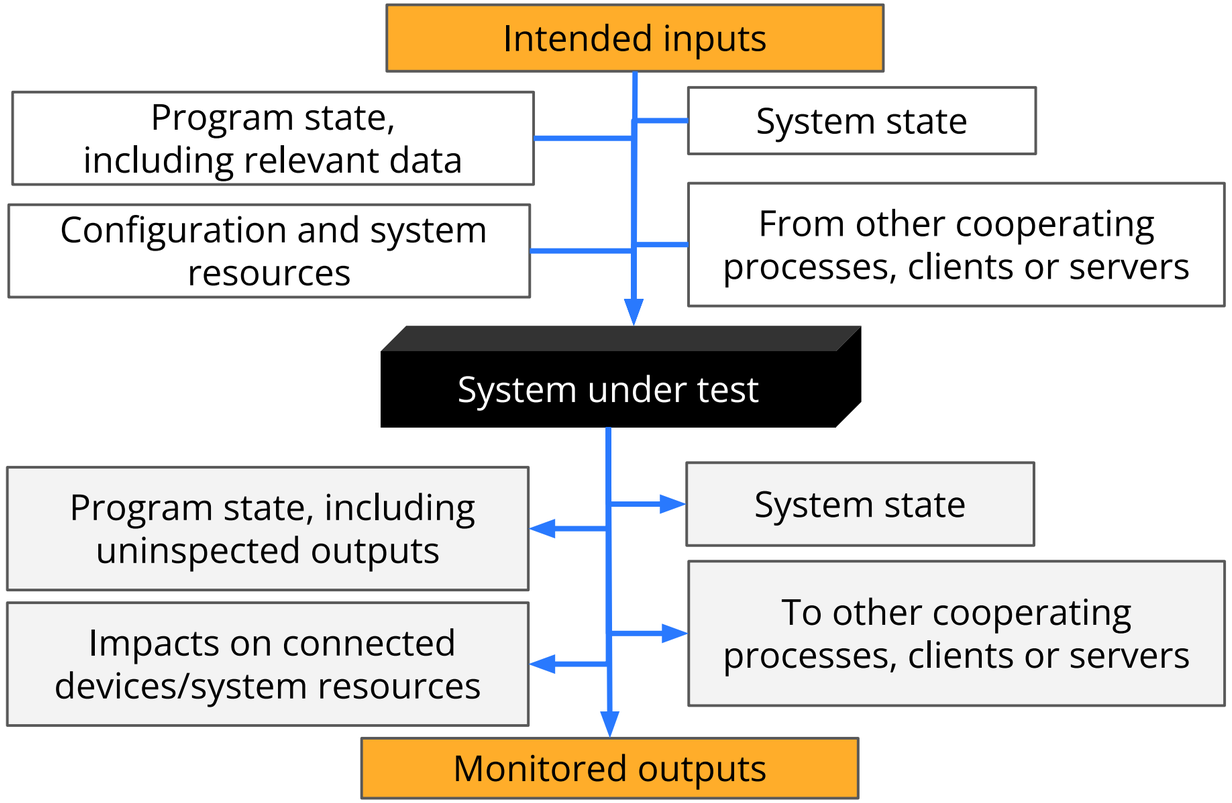
Wouldn’t Test Scripts Solve This?

Test script:

- Detailed test description:
  - Everything you do, step by step
  - “Every” response from the program
  - Every data item
  - Any other relevant setup (e.g. test environment)
- BUT, they tend to be:
  - Enormously expensive
  - Ineffective for finding bugs
- And they only mitigate risk of irreproducibility caused by you forgetting what you actually did

Many Preconditions & Postconditions

Based on Notes from Doug Hoffman



Non-Reproducible Failures

The fact that a bug is not reproducible is information.

- The program is telling you that you have a hole in your logic.
- You are not considering the relevance of (and not checking values of)
    - some of the input conditions (including environmental preconditions), or
    - some of the output conditions
- Therefore, you miss the critical condition(s) needed to reproduce this failure.
- Therefore, you need to consider other conditions.

Use Tools to Capture More Information

Especially if the program (or platform) under test often fails in hard to reproduce ways, use tools. e.g.:

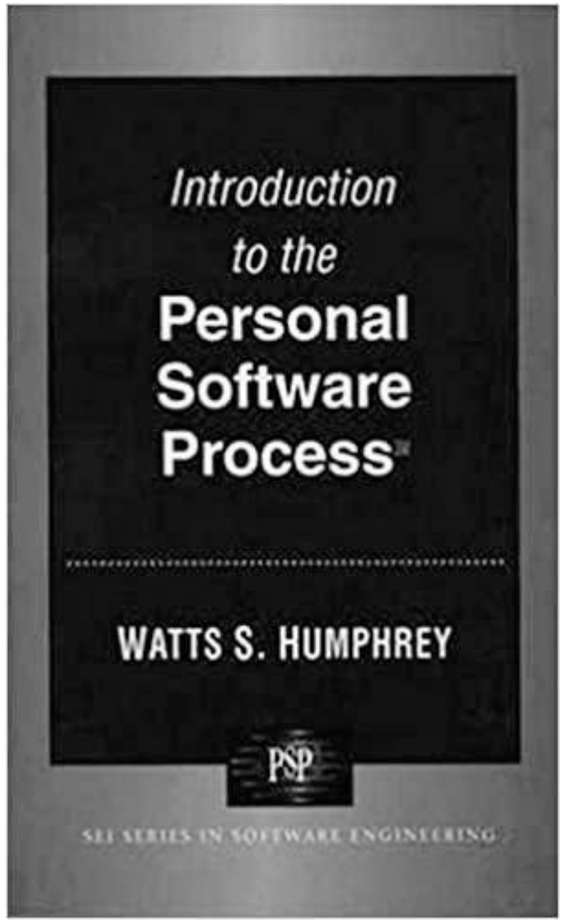
- Capture everything on your screen, in a series of slides that show the sequence of the test:
  - Wink (www.debugmode.com/wink)
  - Camtasia (https://www.techsmith.com/video-editor.html)
  - UserVue (www.techsmith.com)
  - Jing (https://www.techsmith.com/jing-tool.html)
  - FlashBack (https://www.flashbackrecorder.com)
- Lots of sites that link to tools. Here's an example, https://www.techradar.com/news/the-best-free-screen-recorder
- Test with the programmer's debugger loaded.
- Capture state information over time with tools like FileMon, RegMon, Process Monitor, etc. (technet.microsoft.com/en-us/sysinternals/bb795535.aspx)
- In web-based testing, use a proxy. Record all traffic.
  - Fiddler (https://www.telerik.com/fiddler)
  - Charles (https://www.charlesproxy.com)

Non-Reproducible Failures

Watts Humphrey recommends programmers keep a private record of their errors, reviewing them to look for patterns. Programmers have characteristic errors, errors they tend to make over and over. On recognizing one, the programmer can avoid repeating it. Ultimately, the PSP practitioner's bug rate drops a lot.

A non-reproducible bug is a tester's error. Do testers have characteristic blind spots for failure-related conditions? Sometimes, programmers find the cause of non-repro bugs; when they do, they can help you identify the critical condition that you missed. Watts Humphrey suggested to me the idea of testers' tracking this information to discover their blind spots.

To improve over time, keep track of the bugs you're missing and what conditions you are not attending to (or find too hard to manipulate).



Non-Reproducible Errors

The following pages list some conditions commonly ignored or missed by testers.

This is based on personal experience and brainstorming sessions several years ago. It's a bit dated.

Your personal list will be different, but maybe this is a good start. Over time, you can customize this list based on your experiences.

When you run into a irreproducible bug look at this list and ask whether any of these conditions could be the critical one. If it could, vary your tests on that basis and you might reproduce the failure.

Examples of Conditions Often Missed

There are plenty of other conditions that are relevant in your environment.

Start with the ones in this list but add others as you learn of them. How do you learn? Sometimes, someone will fix a bug that you reported as non-reproducible. Call the programmer, ask him how to reproduce it, what are the critical steps that you have to take?

You need to know this anyway, so that you can confirm that a bug fix actually worked.

Examples of Conditions Often Missed

Some problems have delayed effects:

- a memory leak might not show up until after you cut and paste 20 times.
- stack corruption might not turn into a stack overflow until you do the same task many times.
- a wild pointer might not have an easily observable effect until hours after it was mis-set.

If you suspect that you have time-delayed failures, use tools such as videotape, capture programs, debuggers, debug-loggers, or memory meters to record a long series of events over time.

With a time-delayed bug, you won't be able to recreate the failure until you think backwards in time and repeat the tests you did before the tests that apparently caused the problem.

Examples of Conditions Often Missed

- The bug depends on the value of a hidden input variable. (Bob Stahl teaches this well.) In any test, there are the variables we think are relevant and there is everything else. If the data you think are relevant don't help you reproduce the bug, ask what other variables were set, and what their values were.
- Some conditions are hidden; others are invisible. You cannot manipulate them and so it is harder to recognize that they're present. You might have to talk with the programmer about what state variables or flags get set in the course of using a particular feature.
- Some conditions are catalysts. They make failures more likely to be seen. Example: low memory for a leak; slow machine for a race. Some catalysts are more subtle, such as use of one feature that has a subtle interaction with another.

Examples of Conditions Often Missed

BBST ®
BUG ADVOCACY

- Some bugs are predicated on corrupted data. They don't appear unless there are impossible configuration settings in the config files or impossible values in the database. What could you have done earlier today to corrupt this data?
- Programs have various degrees of data coupling. When two modules use the same variable, oddness can happen in the second module after the variable is changed by the first. (Books on structured design, such as Yourdon/Constantine often analyze different types of coupling in programs and discuss strengths and vulnerabilities that these can create.) In some programs, interrupts share data with main routines in ways that cause bugs that will only show up after a specific interrupt.
- The bug depends on you doing related tasks in a specific order.

Examples of Conditions Often Missed

BBST ®
BUG ADVOCACY

- The bug might appear only at a specific time of day or day of the month or year. Look for week-end, month-end, quarter-end and year-end bugs, for example.
- Special cases appear in the code because of time or space optimizations or because the underlying algorithm for a function depends on the specific values fed to the function (talk to your programmer).
- The bug is caused by an error in error-handling. You have to generate a previous error message or bug to set up the program for this one.
- The program may depend on one version of a DLL. A different program loads a different version of the same DLL into memory. Depending on which program is run first, the bug appears or doesn't.

Examples of Conditions Often Missed

BBST ®
BUG ADVOCACY

- The bug is caused by a race condition or other time-dependent event, such as:
  - An interrupt was received at an unexpected time.
  - The program received a message from another device or system at an inappropriate time (e.g. after a time-out.)
  - Data was received or changed at an unexpected time.
- Time-outs trigger a special class of multiprocessing error handling failures. These used to be mainly of interest to real-time applications, but they come up in client/server work and are very pesky:
  - Process A sends a message to Process B and expects a response. B fails to respond. What should A do? What if B responds later?

Examples of Conditions Often Missed

BBST ®
BUG ADVOCACY

- Another inter-process error-handling failure - Process A sends a message to B and expects a response. B sends a response to a different message, or a new message of its own. What does A do?
- The program might be showing an initial state bug, such as:
    - The bug occurs only the first time you run the program (so it happens once on every machine). To recreate the bug, you might have to reinstall the program. If the program doesn't uninstall cleanly, you might have to install on a fresh machine (or restore a copy of your system taken before you installed this software) before you can see the problem.
    - The bug appears once after you load the program but won't appear again until you exit and reload the program.

Examples of Conditions Often Missed

BBST ®
BUG ADVOCACY

- You’re being careful in your attempt to reproduce the bug, and you’re typing too slowly to recreate it.
- The problem depends on a file that you think you’ve thrown away, but it’s actually still in the Trash (where the system can still find it).
- A program was incompletely deleted, or one of the current program’s files was accidentally deleted when that other program was deleted. (Now that you’ve reloaded the program, the problem is gone.)
- The program was installed by being copied from a network drive, and the drive settings were inappropriate or some files were missing. (This is an invalid installation, but it happens on many customer sites.)

Examples of Conditions Often Missed

BBST ®
BUG ADVOCACY

- The bug depends on co-resident software, such as a virus checker or some other process, running in the background. Some programs run in the background to intercept foreground programs' failures. These may sometimes trigger failures (make errors appear more quickly).
- You forgot some of the details of the test you ran, including the critical one(s) or you ran an automated test that lets you see that a crash occurred but doesn't tell you what happened.
- The bug depends on a crash or exit of an associated process.
- The program might appear only under a peak load, and be hard to reproduce because you can't bring the heavily loaded machine under debug control (perhaps it's a customer's system).

Examples of Conditions Often Missed

- The bug occurred because a device that it was attempting to write to or read from was busy or unavailable.
- It might be caused by keyboard keybounce or by other hardware noise.
- On a multi-tasking or multi-user system, look for spikes in background activity.
- The bug is specific to your machine's hardware and system software configuration. (This common problem is hard to track down later, after you've changed something on your machine. That's why good reporting practice involves replicating the bug on a second configuration.)

Examples of Conditions Often Missed

- Code written for a cooperative multitasking system can be thoroughly confused, sometimes, when running on a preemptive multitasking system.
    - (In the cooperative case, the foreground task surrenders control when it is ready. In the preemptive case, the operating system allocates time slices to processes. Control switches automatically when the foreground task has used up its time. The application is suspended until its next time slice. This switch occurs at an arbitrary point in the application's code, and that can cause failures.)

Examples of Conditions Often Missed

- The apparent bug is a side-effect of a hardware failure.
    - For example, a flaky power supply creates irreproducible failures.
    - Another example: one prototype system had a high rate of irreproducible firmware failures. Eventually, these were traced to a problem in the building's air conditioning. The test lab wasn't being cooled, no fan was blowing on the unit under test, and prototype boards in the machine ran very hot. The machine was failing at high temperatures.
- Finally, the old standby:
    - Elves tinkered with your machine when you weren't looking. (For example, at one company, my manager would demonstrate the software to people, **using my machine**, while I was at lunch. If you have to deal with elves, set up a capture program on your system so you'll be able to find out what they did when you come back.)

Close Non-Reproducible Bugs?

BBST ®
BUG ADVOCACY

Problem:

- Non-reproducible bugs burn lots of programmer troubleshooting time.
- Serious problems that are hard to reproduce can be discovered, reported, and closed multiple times without anyone recognizing the pattern. Therefore, it's useful to keep non-repro bugs open and review them for patterns.
- Until they're closed, they show up in open-bug statistics.
- In companies that manage by bug numbers, there is high pressure to close irreproducible bugs quickly.

Throwing Bugs Into the Dumpster

The Dumpster:

- A resolution code that puts the bug into an ignored storage place.
    - The bug shows up as resolved (or is just never counted) in the bug statistics, but it is not closed. It is in a holding pattern.
- Assign a non-reproducible bug to the dumpster if you (programmers and testers)
    - have spent lots of time on it but
    - you don’t think that more work on the bug will be fruitful until more failures provide more information.

Next We Do Dumpster-Diving

BBST ®
BUG ADVOCACY

- Every week or two, (testers and/or programmers) scan the dumpster bugs for similar failures. Sometimes, you'll find several similar reports. If you (or the programmer) think there are enough variations in the reports to provide useful hints on how to reproduce the bug, spend time on the collection. If you (or the programmer) can reproduce the bugs, reopen them and fix or defer them.
- Near the end of the project, do a final review of bugs in the dumpster. These will either close as non-reproducible or be put through one last scrutiny.
- (This is an unusual practical suggestion, but it has worked for clients of mine.)

Summing Up

1. Many reasons for not fixing bugs
2. Non-repro is one of the most common, and most easily defended
3. Few failures are inherently non-reproducible
4. Usually, the critical condition for reproducing the bug is an unusual one, not imagined by the tester
5. Tools help capture information to support troubleshooting
6. Tracking critical conditions for non-repro bugs can improve awareness
7. Multiple reports provide patterns that can help troubleshooting. Keeping multiple reports of non-repro bugs open can help this troubleshooting effort.

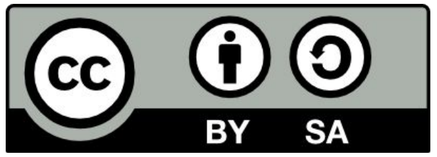
Black Box Software Bug Advocacy
Lecture 5 - Bugs that Could Be Dismissed as Unreasonable, Unrealistic, or Unimportant

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Rebecca L. Fiedler, M.B.A., PH.D.

Retired, President of Kaner, Fiedler & Associates



Course Overview: Fundamental Topics

1. Basic Concepts
2. Effective Advocacy: Making People Want to Fix the Bug
3. Writing Clear Bug Reports
4. Anticipating and Dealing With Objections: Irreproducible Bugs
5. Bugs that Could Be Dismissed as Unreasonable, Unrealistic, or Unimportant
6. Credibility and Influence

Overcoming Objections

Programmers resist spending time on a bug if:

- The programmer can't replicate the failure.
- **The programmer doesn't understand the report.** ←
- **It will take too much work to figure out what the reporter is complaining about and what steps he actually took.** ←
- It seems unrealistic (e.g. "corner case" or requires a complex sequence of improbable steps)
- It is perceived as unimportant:
  No perceived customer impact, minor failure, unused feature.
- It's (allegedly) not a bug, it's a feature.
- It will take a lot of work to fix the bug or will introduce too much risk into the code.
- Management doesn't care about bugs like this.
- The programmer doesn't like/trust you (or the customer who is complaining about the bug).

Overcoming Objections

Programmers resist spending time on a bug if:

- The programmer can't replicate the failure.
- The programmer doesn't understand the report.
- It will take too much work to figure out what the reporter is complaining about and what steps he actually took.
- **It seems unrealistic (e.g. "corner case" or requires a complex sequence of improbable steps)** ←
- **It is perceived as unimportant: No perceived customer impact, minor failure, unused feature.** ←
- It's (allegedly) not a bug, it's a feature.
- It will take a lot of work to fix the bug or will introduce too much risk into the code.
- Management doesn't care about bugs like this.
- The programmer doesn't like/trust you (or the customer who is complaining about the bug).

The Unrealistic Failure

We saw this in Lesson 2 (corner cases).

1. use extreme tests to expose bugs
2. then modify the tests to expose bugs under less extreme conditions (or demonstrate that the bugs really are restricted to the extremes)

If the problem DOES only occur in extreme cases:

- How many people might actually run into this extreme? (Get data)
- How serious is the failure?

If you can't do the follow-ups:

- Ask for causal analysis before dismissal

No Customer Impact: Old Bug

- The bug has been in the product for several versions, no one (allegedly) cares about it.
  - Work with the technical support or help desk manager to check whether no one cares.
  - Outbound support studies are particularly interesting.

No Customer Impact: Generally

If your report of some other type of bug or design issue is dismissed as having "no customer impact," ask yourself:

Hey, how do they know the customer impact?

Then check with people who might actually have data or experience:

- Technical marketing
- Human factors
- Network admins
- In-house power users
- Technical support
- Documentation
- Training
- Maybe sales

The stakeholders who are most affected by a bug should explain the bug's costs.

Overcoming Objections

Programmers resist spending time on a bug if:

- The programmer can't replicate the failure.
- The programmer doesn't understand the report.
- It will take too much work to figure out what the reporter is complaining about and what steps he actually took.
- It seems unrealistic (e.g. "corner case" or requires a complex sequence of improbable steps)
- It is perceived as unimportant: No perceived customer impact, minor failure, unused feature.
- **It's (allegedly) not a bug, it's a feature.** ←
- It will take a lot of work to fix the bug or will introduce too much risk into the code.
- Management doesn't care about bugs like this.
- The programmer doesn't like/trust you (or the customer who is complaining about the bug).

It’s (Allegedly) Not a Bug. It’s a Feature

- “**An argument over whether something is a bug**”
  is really
  “**an argument about the oracle you should use**”
  to evaluate your test results.

Use Oracles to Resolve Arguments

BBST ®
BUG ADVOCACY

An oracle is the principle or mechanism by which you recognize a problem.

- **"...it works"**

really means

- **"...it appeared to meet some requirement to some degree."**

Use Oracles to Resolve Arguments

BBST ®
BUG ADVOCACY

An oracle is the principle or mechanism by which you recognize a problem.

- **“…it doesn’t work”**

often means

- **“…it violates my expectations.”**

The Specification Oracle

Suppose the program says “It’s not a bug because the program meets the specification.”

"Meets the specification" is a consistency oracle:

- The program might work as specified but offer low value to the stakeholder
- The program DOESN’T work as specified because it
  - works better than the specification
  - handles a situation not anticipated in the specification
  - takes into account stakeholder needs that were missed in the specification.

A program that exactly meets a specification is correct if and only if the specification is complete and correct

Remember Those Consistency Oracles

Consistent within product	Function behavior consistent with behavior of comparable functions or functional patterns within the product
Consistent with comparable products	Function behavior consistent with that of similar functions in comparable products
Consistent with history	Present behavior consistent with past behavior
Consistent with our image	Behavior consistent with an image the organization wants to project
Consistent with claims	Behavior consistent with documentation, specifications, or ads
Consistent with standards or regulations	Behavior consistent with externally-imposed requirements
Consistent with user's expectations	Behavior consistent with what we think users want
Consistent with purpose	Behavior consistent with product or function's apparent purpose

These are especially useful for explaining a bug and its significance.

It’s Not a Bug, It’s a Feature (?)

BBST ®
BUG ADVOCACY

To argue that the program is misbehaving.

- cite one or more of the consistency heuristics
- show that the heuristic(s) would predict that the program would behave differently
- explain how the program would behave if it conformed to these heuristics
- if possible, tie the difference to potential impact on a stakeholder

Overcoming Objections

Programmers resist spending time on a bug if:

- The programmer can't replicate the failure.
- The programmer doesn't understand the report.
- It will take too much work to figure out what the reporter is complaining about and what steps he actually took.
- It seems unrealistic (e.g. "corner case" or requires a complex sequence of improbable steps)
- It is perceived as unimportant: No perceived customer impact, minor failure, unused feature.
- It's (allegedly) not a bug, it's a feature.
- It will take a lot of work to fix the bug or will introduce too much risk into the code.
- **Management doesn't care about bugs like this.** ←
- The programmer doesn't like/trust you (or the customer who is complaining about the bug).

You Shouldn’t Report These Bugs?

BBST ®
BUG ADVOCACY

- Report a design issue, it gets rejected. Report essentially the same issue—at what point are you just wasting time?
- Contract-based development, customer rejected some obvious improvements to the product. Should you report them, given that the customer will not pay for them?
- Late in the project, high fear of side-effects. Straightforwardly minor coding errors (spelling mistakes in help text) won’t be fixed. Should you report them?
- Alternatives:
    - MIP (mention in passing)
    - Alternate database

Denial of service attack on a project: report lots of deferrable bugs late in development.

Overcoming Objections

BBST ®
BUG ADVOCACY

Programmers resist spending time on a bug if:

- The programmer can't replicate the failure.
- The programmer doesn't understand the report.
- It will take too much work to figure out what the reporter is complaining about and what steps he actually took.
- It seems unrealistic (e.g. "corner case" or requires a complex sequence of improbable steps)
- It is perceived as unimportant: No perceived customer impact, minor failure, unused feature.
- It's (allegedly) not a bug, it's a feature.
- It will take a lot of work to fix the bug or will introduce too much risk into the code.
- Management doesn't care about bugs like this.
- **The programmer doesn't like/trust you (or the customer who is complaining about the bug).** ←

Summing Up

Programmers and project managers reject bugs for a lot of reasons.

Some of these sound unreasonable on their face, however:

1. Sometimes, they are not unreasonable
2. You can have a big impact on whether one of these types of rejections happens, or is accepted by the triage team,
    - by the wording of your reports, and
    - the research you provide beyond the specific reproduce-the-bug details.

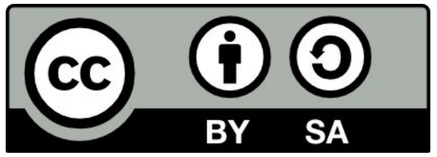
Black Box Software Bug Advocacy
Lecture 6
Credibility and Influence

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Rebecca L. Fiedler, M.B.A., PH.D.

Retired, President of Kaner, Fiedler & Associates



Course Overview: Fundamental Topics

1. Basic Concepts
2. Effective Advocacy: Making People Want to Fix the Bug
3. Writing Clear Bug Reports
4. Anticipating and Dealing With Objections: Irreproducible Bugs
5. Bugs that Could Be Dismissed as Unreasonable, Unrealistic, or Unimportant
6. Credibility and Influence

Credibility and Influence

Some people are more successful at getting the bugs they report fixed than others. Why?

- One factor is the technical quality of the report (follow-up testing, impact analysis, clarity of writing, etc.)
- A different factor is the extent to which the reporter is credible or influential with the people who are evaluating the report.

How do you build (or lose) that influence?

Your Choices and Your Credibility

BBST ®
BUG ADVOCACY

What happens to your reputation if you:

- Report every bug, no matter how minor, to make sure no bug is ever missed?
- Report only the serious problems (the “good bugs”)?
- Fully analyze each bug?
- Only lightly analyze bugs?
- Insist that every bug get fixed?

Your decisions reflect on your judgment and will cumulatively affect your credibility.

The comprehensibility of your reports and the extent and skill of your analysis will also have a substantial impact on your credibility.

See Kaner, C., Falk, J., & Nguyen, H.Q., (2nd Edition, 2000), *Testing Computer Software*, pages 90-97, 115-118

Managing a Series of Decisions

Bug handling involves a series of many decisions by different people, such as:

Tester:

- Should I report this bug?
- Should I report these similar bugs as one bug or many?
- Should I report this awkwardness in the user interface?
- Should I stop reporting bugs that look minor?
- How much time should I spend on analysis and styling of this report?

Managing a Series of Decisions

BBST ®
BUG ADVOCACY

Programmer:

- Should I fix this bug or defer it?

Project Manager:

- Should I defer this bug?
- If the programmer wants to defer this bug, should I approve that?
- Do I know what the underlying problem is? Does the programmer? Do we know if there are other risks? Do I know how long this will take to fix? Do I know what decision to make? Do I trust the programmer's intuition or judgment?

Managing a Series of Decisions

BBST ®
BUG ADVOCACY

Tester:

- Should I appeal the deferral of this bug?
- How much time should I spend analyzing this bug further?

One critical lesson:

If you're going to fight, win.

Managing a Series of Decisions

Test Group Manager (or Test Lead):

- Should I make an issue about this bug?
  - If the project team won't fix it, should I raise it with more senior management?
  - Is my staff being treated with appropriate respect in this process?
  - Was this deferred because my staff is writing weak bug reports?
- Should I encourage my tester to
  - investigate the bug further?
  - argue the bug further?
  - or to quit worrying about this one?
  - or should I just keep out of the discussion this time?

Managing a Series of Decisions

Customer Service, Marketing, Documentation:

- Should I ask the project manager to reopen this bug?
- (The tester appealed the deferral) Should I support the tester this time?
- Should I spend time trying to figure this thing out?
- Will this require extra work on the manual/support notes/ advertising/help?

Director, Vice President, other senior staff:

- Should I override the project manager's deferral of this bug?

Modeling Decision-Making

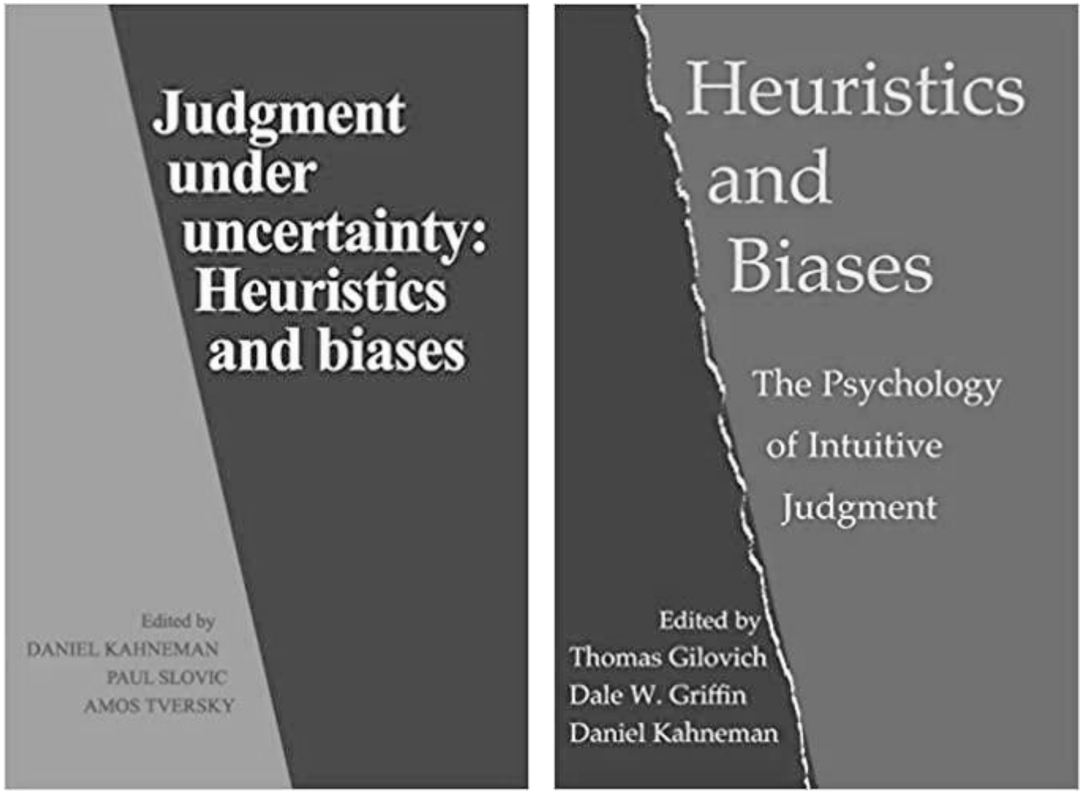
Bug reporting/fixing involves a series of decisions:

- made under time pressure
- with incomplete information
- and incomplete knowledge of consequences.

Some decisions will be wrong.

Decision-making under uncertainty is influenced by:

- the available information
- the decision-maker's heuristics
- the decision-maker's biases



Decisions Are Subject to Bias

- Much of this is unconscious.
- The bias testers are most familiar with in testing is preferred result:
  - Glen Myers (Art of Software Testing): testers who want to find bugs are more likely to notice program misbehavior than people who want to verify that the program works correctly.
  - Under the name Experimenter Effects, this phenomenon has been widely studied and confirmed across the fields of science.
  - If you WANT an experiment to come out one way instead of another, you are more likely to get that result. That is, you are likely to bias how you design the experiment, how you run the experiment, how you deal with your mistakes, how you analyze the data, how you read the graphs—to yield what you are looking for.

For MUCH more on this, search the phrase "experimenter effects" or for writings by Robert Rosenthal

Decisions Are Subject to Bias

Prime biasing variables are:

- Motivation (preferred result)
- Perceived probability:
  - If you think that an event is unlikely, you will be substantially less likely (below the actual probability) to report it.
- Expected consequence of a decision:
  - What happens if you make a False Alarm? Is this worse than a Miss or less serious?
- Perceived importance of the task:
  - A person who perceives a decision as very important may be much more likely to rely on careful observation and less likely to respond randomly or primarily on their biases...

Since the 1950's, application of signal detection theory to human perception has had a profound influence on the nature and quality of models of subjective experience.

Signal Detection & Recognition

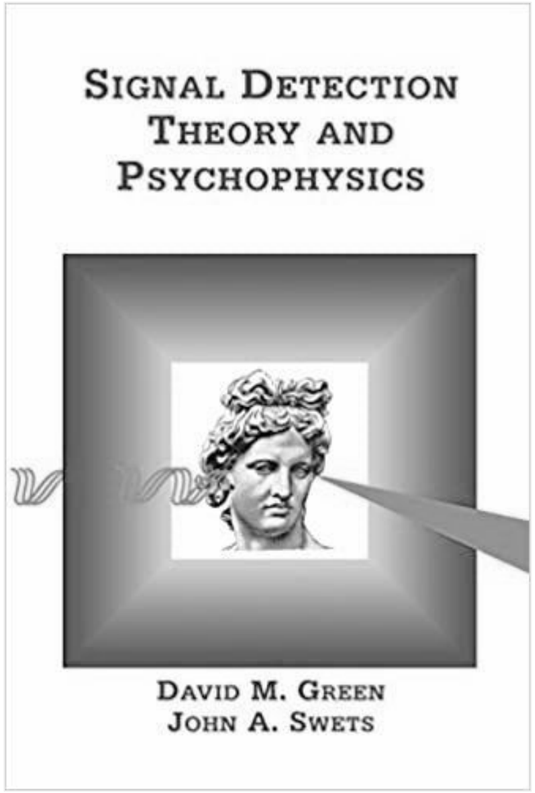
- **Hit**: The signal (the event you are looking for) is present and you detect (recognize) it. In the testing case, you are looking for a bug, a software failure.
- **Miss**: The signal (the program fails) is present, but you don't recognize it
- **False Alarm**: The signal is not present (the program works OK), but you report the signal (write a bug report)
- **Correct Rejection**: The signal is not present and you don't report it as present

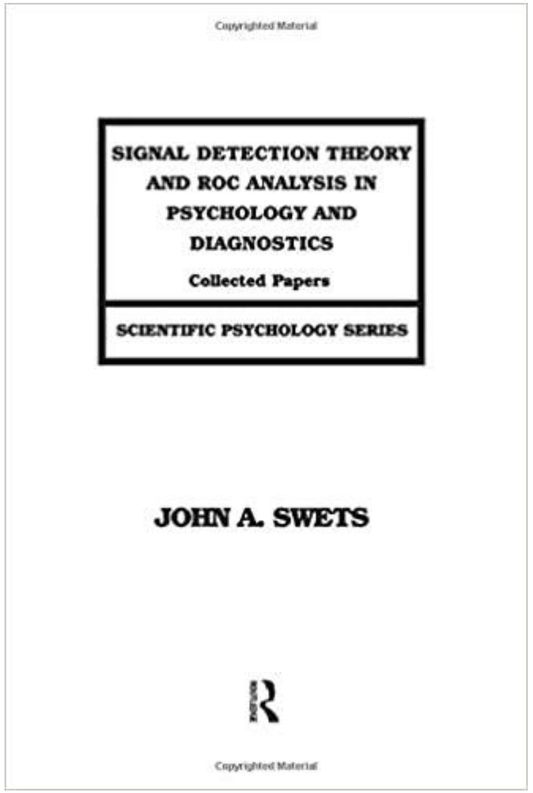
		Response	
		Bug	**Feature**
Actual Event	**Bug**	Hit	Miss
	Feature	False alarm	Correct Rejection

Making Decisions Under Uncertainty

We make decisions with incomplete information and **therefore** we make mistakes.

- How can we reduce the number of false alarms without increasing the number of misses?
- How can we increase the number of hits without increasing the number of false alarms?
- Pushing people to make fewer of one type of reporting error will **inevitably** result in an increase in another type of reporting error.
- Training, specs, etc. help, but the basic problem remains.





Decisions Have Consequences

Test groups establish policies and procedures for handling bugs, as do the groups who receive and evaluate the bug reports.

Those policies affect how the bugs are treated, both directly (that's what policies DO) and indirectly (by creating biases)

Bug reporting policymakers must consider the effects on the overall decision-making system, not just on the tester and first-level bug reader.

To Improve the Quality of Reports

- Give people the time needed to do good work and show that you expect good work
- Give feedback to reporters that explains why their reports were rejected:
  - weak communication quality
  - weak analysis
  - weak underlying problem
  - exaggerated or incredible assessment of consequences.
- Explain the costs of spurious reports:
  - processing time
  - opportunity cost

If you want better reports, use collegial feedback to train the reporter.

Biasing People Who Report Bugs

To increase probability that people report bugs:

- Encourage testers to report all anomalies.
- Praise well-written reports.
- Create contests to recognize best bug reports.
- Publish feedback about the impact of bug fixes (estimated savings in support costs).
- Give feedback to non-testers who report bugs about the results of their reports.
- Adopt a formal system for challenging bug deferrals.
- Give positive feedback (“keep up the good work”) when a genuine bug is reported, but deferred.

Biasing People Who Report Bugs

To increase probability that people report bugs:

- Manage the friction that comes from bugs hitting the system at hard-to-handle times. Weigh schedule urgency consequences against an appraisal of quality costs.
    - Early in the schedule, report all types of bugs
    - late in the schedule,
        - be more selective about challenging the design, identifying desirable additional benefits, or reporting low-visibility minor problems.
        - allow for informal reporting processes so that serious bugs can get noticed quickly (MIP-mention in passing).
        - set up a separate database for design/minor issues (which will be evaluated for the start of the next release).

Biasing People Who Report Bugs

To reduce valid bugs:

- convince them their work is pointless or will be ignored,
- make them feel unappreciated,
- make them feel as though they aren't team players or they have a bad attitude
- treat every report as a personal attack,
- emphasize the urgent need for this product in the field
- create incentives for not reporting bugs, or
- create incentives for other people to pressure them not to report bugs.

Bias-Risky Conduct

- **Don’t use bug statistics for employee bonus or discipline.** ←
- **Don’t use bug stats to embarrass people.** ←
- Be very cautious about filtering reports of “bugs” you consider minor (or features). Refusing to allow someone else’s reports into the bug database discourages them.
- Rarely change an in-house bug reporter’s language without their free permission. Show respect for their words. Add your comments as additional notes.
- Monitor language in the reports that is critical of the programmer or the tester.
- Don’t uncritically accept someone’s dismissal of a tester’s report as very unlikely in the field.

Bias-Risky Conduct

- Don’t use bug statistics for employee bonus or discipline.
- Don’t use bug stats to embarrass people.
- **Be very cautious about filtering reports of “bugs” you consider minor (or features). Refusing to allow someone else’s reports into the bug database discourages them.** ←
- **Rarely change an in-house bug reporter’s language without their free permission. Show respect for their words. Add your comments as additional notes.** ←
- Monitor language in the reports that is critical of the programmer or the tester.
- Don’t uncritically accept someone’s dismissal of a tester’s report as very unlikely in the field.

Bias-Risky Conduct

- Don’t use bug statistics for employee bonus or discipline.
- Don’t use bug stats to embarrass people.
- Be very cautious about filtering reports of “bugs” you consider minor (or features). Refusing to allow someone else’s reports into the bug database discourages them.
- Rarely change an in-house bug reporter’s language without their free permission. Show respect for their words. Add your comments as additional notes.
- **Monitor language in the reports that is critical of the programmer or the tester.** ←
- **Don’t uncritically accept someone’s dismissal of a tester’s report as very unlikely in the field.** ←

Biasing People Who Evaluate Bug Reports

These reduce the probability that the bug will be taken seriously and fixed.

- Language critical of the programmer
- Inflated estimate of the bug's severity
- Pestering & refusing to ever take "No" for an answer
- Incomprehensibility, excessive detail, or apparent narrowness of the report
- Tight schedule (no time to fix anything)
- Weak reputation of the reporter
- Management encouragement to ignore/defer bugs
- Contempt for the end user or customer

Biasing People Who Evaluate Bug Reports

These increase the probability that a bug will be taken seriously and fixed.

- It conflicts with reliability or regulatory requirements in this market, or violates the contract
- Persuasive real-world impact
- Reported by customer/beta (who says it's important) rather than internally
- Strong reputation of the reporter
- Weak reputation of the programmer who created or deferred the bug
- Poor quality/performance compared to competitive product(s)
- News of litigation in the press

Clarify Expectations

BBST ®
BUG ADVOCACY

An important test management task is to facilitate understanding and agreement about the policies and procedures for bug reporting/tracking.

- Track open issues/tasks or just bugs?
- Track documentation issues or just code?
- Track minor issues late in the schedule or not?
- Track issues outside of the published spec and requirements or not?
- How to deal with similarity?

Making the rules explicit helps prevent misunderstandings about what's personal and what's not.

Summing Up

1. Some people are more successful at getting bugs fixed. Keeping in mind the subjective factors in assessing bug reports, they:
    - Write good reports (clear writing, good troubleshooting)
    - Report significant problems
    - Never exaggerate the severity of their bug
    - Provide good reasons/data for arguing that a bug should be fixed, or admit that they are speaking from intuition
    - Treat the programmers and other stakeholders with courtesy and respect (even—especially—ones who don't deserve it.)
2. If they respect you and like you, they'll treat your bug reports with more respect.

Summing Up the Course

1. Quality is subjective
2. Bugs are threats to the value of the product to a stakeholder. I normally report anything that any stakeholder with influence might want fixed. However, if my testing mission is tightly focused, I operate within the mission. For example, if I am doing pre-alpha testing with the programmers, helping them do a very early review of their code, I might work with them only on certain types of coding errors because that might be the only service that will help them write better code at that time.
3. A bug report will have a lot more effect if it ties the problem being reported to the loss of value to the stakeholder with influence.
4. In general, bug reporting is a persuasive activity. We are exercising influence, not just writing down value-neutral technical details.

Summing Up the Course

5. We can simplify the persuasive task by saying that we are trying to
    - motivate people to fix our bugs and
    - overcome their objections to fixing those bugs.

    We have many methods for improving reports in both ways.
6. A failure is a misbehavior caused by an underlying error. An error can cause many different failures. We should look for the worst failure and focus reporting on that.
7. The entire bug-handling process involves a series of decisions that are heavily influenced by the decision-makers' preconceived notions—their heuristics and biases—about whose reports are worthwhile and what problems are likely to be important. The credibility of the tester has a big impact on these decisions.