Black Box Software Testing (Professional Seminar)

Cem Kaner, J.D., Ph.D.

Professor of Computer Sciences
Florida Institute of Technology

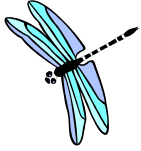### Section:7
Bug Advocacy

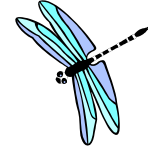Summer, 2002
Contact Information:
kaner@kaner.com
www.kaner.com (testing website)
www.badsoftware.com (legal website)

Bug Advocacy:

How to
Win Friends,

```
influence programmers
```
and

SToMp BUGs.

(Not necessarily in that order.)

Bug Advocacy?

1. The point of testing is to find bugs.

2. **Bug reports are your primary work product.** This is what people outside of the testing group will most notice and most remember of your work.

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

4. Programmers operate under time constraints and competing priorities. For example, outside of the 8-hour workday, some programmers prefer sleeping and watching Star Wars to fixing bugs.

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

Note: When I say "the best tester is the one who gets the most bugs fixed," I am not encouraging bug counting metrics, which are almost always counterproductive. Instead, what I am suggesting is that the effective tester looks to the effect of the bug report, and tries to write it in a way that gives each bug its best chance of being fixed. Also, a bug report is successful if it enables an informed business decision. Sometimes, the best decision is to not fix the bug. The excellent bug report raises the issue and provides sufficient data for a good decision.

Selling Bugs

Time is in short supply. If you want to convince the programmer to spend his time fixing your bug, you may have to sell him on it.

> *(Your bug? How can it be your bug? The programmer made it, not you, right? It's the programmer's bug.* Well, yes, but you found it so now it's yours *too.)*

Sales revolves around two fundamental objectives:

- **Motivate the buyer (*Make him WANT to fix the bug.*)**

- **Overcome objections (*Get past his excuses and reasons for not fixing the bug.*)**

Motivating the Bug Fixer

Some things that will often make programmers want to fix the bug:

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

Overcoming Objections

These make programmers resist spending time on a bug:

- The programmer can't replicate the defect.
- Strange and complex set of steps required to induce the failure.
- Not enough information to know what steps are required, and it will take a lot of work to figure them out.
- The programmer doesn't understand the report.
- Unrealistic (e.g. "corner case")
- It will take a lot of work to fix the defect.
- A fix will introduce too much risk into the code.
- No perceived customer impact
- Unimportant (no one will care if this is wrong: minor error or unused feature.)
- That's not a bug, it's a 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)

Bug Advocacy
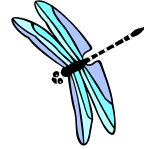
Motivating Bug Fixes

By Better Researching

The Failure Conditions
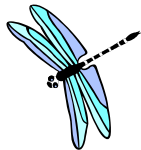
Motivating The Bug Fix:
Looking At The Failure

Some vocabulary

- An ***error*** (or ***fault***) is a design flaw or a deviation from a desired or intended state.

- An error won't yield a failure without the ***conditions*** that trigger it. Example, if the program yields 2+2=5 on the 10th time you use it, you won't see the error before or after the 10th use.

- The ***failure*** is the program's actual incorrect or missing behavior under the error-triggering conditions.

- A ***symptom*** might be a characteristic of a failure that helps you recognize that the program has failed.

- ***Defect*** is frequently used to refer to the failure or to the underlying error.

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

Motivating The Bug Fix: Looking At The Failure

VOCABULARY EXAMPLE

Here's a defective program

- *INPUT A*
- *INPUT B*
- *PRINT A/B*

What is the fault?

What is the critical condition?

What will we see as the failure?

Motivating the Bug Fix

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

Motivating the Bug Fix: Make it More Serious

LOOK FOR FOLLOW-UP ERRORS

When you find a coding error, you have the program in a state that the programmer did not intend and probably did not expect. There might also be data with supposedly impossible values.

The program is now in a vulnerable state. Keep testing it and you might find that the real impact of the underlying fault is a much worse failure, such as a system crash or corrupted data.

I do three types of follow-up testing:

- ***Vary my behavior*** (change the conditions by changing what I do)
- ***Vary the options and settings of the program*** (change the conditions by changing something about the program under test).
- ***Vary the*** software and hardware ***environment***.

Follow-Up: Vary Your Behavior

Keep using the program after you see the problem.

- **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?**

- **Try things that are _related to the task_ that failed. For example, if the program unexpectedly but slightly scrolls the display when you add two numbers, try tests that affect adding or that affect the numbers. Do X, see the scroll. Do Y then do X, see the scroll. Do Z, then do X, see the scroll, etc. (If the scrolling gets worse or better in one of these tests, follow that up, you're getting useful information for debugging.)**

- **Try things that are _related to the failure_. If the failure is unexpected scrolling after adding, try scrolling first, then adding. Try repainting the screen, then adding. Try resizing the display of the numbers, then adding.**

- **Try entering the numbers more quickly or _changing the speed_ of your activity in some other way.**

- **And _try the usual exploratory testing techniques_. So, for example, you might try some interference tests. Stop the program or pause it 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?**

Follow-Up: Vary Options and Settings

In this case, the steps to achieve the failure are taken as given. Try to reproduce the bug when the program is in a different state:

- Use a different database.
- Change the values of persistent variables.
- Change how the program uses memory.
- Change anything that looks like it might be relevant that allows you to change as an option.

For example, suppose the program scrolls unexpectedly when you add two numbers. Maybe you can change the size of the program window, or the precision (or displayed number of digits) of the numbers, or background the activity of the spell checker.

Follow-Up: Vary the Configuration

A bug might show a more serious failure if you run the program with less memory, a higher resolution printer, more (or fewer) device interrupts coming in etc.

- If there is anything involving timing, use a really slow (or very fast) computer, link, modem or printer, etc..
- If there is a video problem, try other resolutions on the video card. Try displaying MUCH more (less) complex images.

Note that we are not:

- checking standard configurations
- asking how broad the circumstances that produces the bug.

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

Returning to the example (unexpected scrolling when you add two numbers), try things like:

- Different video resolutions
- Different mouse settings if you have a wheel mouse that does semi-automated scrolling
- An NTSC (television) signal output instead of a traditional (XGA or SVGA, etc.) monitor output.

Follow-up: Bug New to This Version?

In many projects, an old bug (from a previous shipping release of the program) might not be taken very seriously if there weren't lots of customer complaints.

- (If you know it's an old bug, check its complaint history.)

- The bug will be taken more seriously if it is new.

- You can argue that it should be treated as new if you can find a new variation or a new symptom that didn't exist in the previous release. What you are showing is that the new version's code interacts with this error in new ways. That's a new problem.

- This type of follow-up testing is especially important during a maintenance release that is just getting rid of a few bugs. Bugs won't be fixed unless they were (a) scheduled to be fixed because they are critical or (b) new side effects of the new bug fixing code.

Motivating the Bug Fix: Show it is More General

LOOK FOR CONFIGURATION DEPENDENCE

Bugs that don't fail on the programmer's machine are much less credible (to that programmer). If they are configuration dependent, the report will be much more credible if it identifies the configuration dependence directly (and so the programmer starts out with the expectation that it won't fail on all machines.)

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

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

LOOK FOR CONFIGURATION DEPENDENCE

In the ideal case (standard in many companies), test on 2 machines

- Do your main testing on Machine 1. Maybe this is your powerhouse: latest processor, newest updates to the operating system, fancy printer, video card, USB devices, huge hard disk, lots of RAM, cable modem, etc.

- When you find a defect, use Machine 1 as your bug reporting machine and replicate on Machine 2. Machine 2 is totally different. Different processor, different keyboard and keyboard driver, different video, barely enough RAM, slow, small hard drive, dial-up connection with a link that makes turtles look fast.

- Some people do their main testing on the turtle and use the power machine for replication.

- Write the steps, one by one, on the bug form at Machine 1. As you write them, try them on Machine 2. If you get the same failure, you've checked your bug report while you wrote it. (A valuable thing to do.)

- If you don't get the same failure, you have a configuration dependent bug. Time to do troubleshooting. But at least you know that you have to.

AS A MATTER OF GENERAL GOOD PRACTICE, IT PAYS TO REPLICATE EVERY BUG ON A SECOND MACHINE.

Motivating the Bug Fix: Show it is More General

UNCORNER YOUR CORNER CASES

We test at extreme values because these are the most likely places to show a defect. *But once we find the defect, we don't have to stick with extreme value tests.*

- Try mainstream values. These are easy settings that should pose no problem to the program. Do you replicate the bug? If yes, write it up, referring primarily to these mainstream settings. This will be a very credible bug report.

Motivating the Bug Fix: Show it is More General

UNCORNER YOUR CORNER CASES

- If the mainstream values don't yield failure, but the extremes do, then do some troubleshooting around the extremes.

    » Is the bug tied to a single setting (a true corner case)?

    » Or is there a small range of cases? What is it?

    » In your report, identify the narrow range that yields failures. The range might be so narrow that the bug gets deferred. That might be the right decision. In some companies, the product has several hundred open bugs a few weeks before shipping. They have to decide which 300 to fix (the rest will be deferred). Your reports help the company choose the right 300 bugs to fix, and help people size the risks associated with the remaining ones.

19
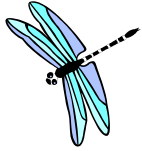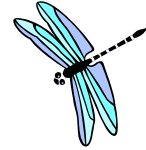
Overcoming
OBJECTIONS
By Better Researching
The Failure Conditions

Overcoming Objections: Analysis of the Failure

Things that will make programmers resist spending their time on the bug:

- ## The programmer can't replicate the defect.

- Strange and complex set of steps required to induce the failure.

- Not enough information to know what steps are required, and it will take a lot of work to figure them out.

- The programmer doesn't understand the report.

- Unrealistic (e.g. "corner case")

- It's a feature.

Objection, Objection: Non-Reproducible Errors

Always report non-reproducible errors. If you report them well, programmers can often figure out the underlying problem.

You must describe the failure as precisely as possible. If you can identify a display or a message well enough, the programmer can often identify a specific point in the code that the failure had to pass through.

- When you realize that you can't reproduce the 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.

- Find ways to affect timing of the program or devices, Slow down, speed up.

- ***Talk to the programmer*** and/or read the code.

Non-Reproducible Errors

- **The fact that a bug is not reproducible is data. The program is telling you that you have a hole in your logic. You are not entertaining certain relevant conditions. Why not?**

- **See Watts Humphrey, Personal Software Process, for recommendations to programmers of a system for discovering and then eliminating characteristic errors from their code. A non-reproducible bug is a tester's error, just like a design bug is a programmer's error. It's valuable to develop a system for discovering your 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).**

- **The following pages give a list of some conditions commonly ignored or missed by testers. Your personal list will be different in some ways, but maybe this is a good start. When you run into a irreproducible defect 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.**

(Note: Watts Humphrey suggested to me the idea of keeping a list of commonly missed conditions. It has been a valuable idea.)

Non-Reproducible Errors: 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.

Non-Reproducible Errors: Examples of Conditions Often Missed

- I highlighted the first three in lecture because so many people have trouble with time-delayed bugs. Until you think backwards in time and ask how you could find a defect that has a delayed reaction effect, you won't be able to easily recreate these problems.

- The following pages give additional examples. There are plenty of other conditions that are relevant in your environment. Start with these 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.

Non-Reproducible Errors:
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 that 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 and 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. But sometimes catalysts are more subtle, such as use of one feature that has a subtle interaction with another.**

- **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?**

Non-Reproducible Errors:
Examples of Conditions Often Missed

- **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.**

- **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.**

- **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 depends on you doing related tasks in a specific order.**

Non-Reproducible Errors:
Examples of Conditions Often Missed

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.

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.

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?

Non-Reproducible Errors:
Examples of Conditions Often Missed

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?

You're being careful in your attempt to reproduce the bug, and you're typing too slowly to recreate it.

The program might be showing an initial state bug, such as:

- The bug appears only the first time after you install the program (so it happens once on every machine.)

- The bug appears once after you load the program but won't appear again until you exit and reload the program.

    - *(See Testing Computer Software's Appendix's discussion of initial state bugs.)*

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.

29

Non-Reproducible Errors: Examples of Conditions Often Missed

- 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.)

- 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).

Non-Reproducible Errors:
Examples of Conditions Often Missed

- 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).
- On a multi-tasking or multi-user system, look for spikes in background activity.
- 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.

Non-Reproducible Errors:
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.**

- **The bug occurs only the first time you run the program or the first time you do a task after booting the program. 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.**

Non-Reproducible Errors:
Examples of Conditions Often Missed

- **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.)**

- **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.**

- **Elves tinkered with your machine when you weren't looking.**

- **There are several other ideas (focused on web testing) at http://www.logigear.com/whats_new.html#article**

Putting Bugs in the Dumpster

Problem:

- Non-reproducible bugs burn a huge amount of programmer troubleshooting time, then get closed (usually abandoned). Until they're closed, they show up in open-bug statistics. In companies that manage more by bug numbers than by good sense, there is tremendous pressure to close irreproducible bugs quickly.

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 whenever you (testers and programmers) spend enough time on it that you don't think that more work on the bug will be fruitful at this time.

Dumpster Diving:

- Every week or two, (testers and/or programmers) go through the dumpster bugs looking for similar failures. At some point, you'll find a collection of several similar reports. If you (or the programmer) think there are enough variations in the reports to provide useful hints on how to repro the bug, spend time on the collection. If you (or the programmer) can repro the bugs, reopen them with the extra info (status is now open, resolution is pending)
- Near the end of the project, do a final review of bugs in the dumpster. These will either close non-repro or be put through one last scrutiny

(This is an unusual practical suggestion, but it has worked for clients of mine.)

# *Overcoming Objections: Analysis of the Failure*

Things that will make programmers resist spending their time on the bug:

- The programmer can't replicate the defect.

- Strange and complex set of steps required to induce the failure.

- Not enough information to know what steps are required, and it will take a lot of work to figure them out.

- The programmer doesn't understand the report.

- Unrealistic (e.g. "corner case")

- It's a feature!
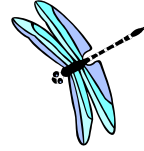
Writing the Bug Report
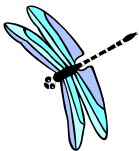
Reporting Errors

As soon as you run into a problem in the software, fill out a Problem Report form. In the well written report, you:

- Explain how to reproduce the problem.

- Analyze the error so you can describe it in a minimum number of steps.

- Include all the steps.

- Make the report easy to understand.

- Keep your tone neutral and non-antagonistic.

- Keep it simple: one bug per report.

- If a sample test file is essential to reproducing a problem, reference it and attach the test file.

- To the extent that you have time, describe the dimensions of the bug and characterize it. Describe what events are and are not relevant to the bug. And what the results are (any characteristics of the failure) and how they varied across tests.

The Problem Report Form

A typical form includes many of the following fields

- **Problem report number: must be unique**
- **Reported by: original reporter's name. Some forms add an editor's name.**
- **Date reported: date of initial report**
- **Program (or component) name: the visible item under test**
- **Release number: like Release 2.0**
- **Version (build) identifier: like version C or version 20000802a**
- **Configuration(s): h/w and s/w configs under which the bug was found and replicated**
- **Report type: e.g. coding error, design issue, documentation mismatch, suggestion, query**
- **Can reproduce: yes / no / sometimes / unknown. (Unknown can arise, for example, when the repro configuration is at a customer site and not available to the lab.)**

The Problem Report Form

A typical form includes many of the following fields

- **Severity: assigned by tester. Some variation on small / medium / large**
- **Priority: assigned by programmer/project manager**
- **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)**
- **Problem summary: 1-line summary of the problem**
- **Key words: use these for searching later, anyone can add to key words at any time**
- **Problem description and how to reproduce it: step by step repro description**
- **Suggested fix: leave it blank unless you have something useful to say**
- **Assigned to: typically used by project manager to identify who has responsibility for researching/fixing the problem**

The Problem Report Form

A typical form includes many of the following fields

- **Status: Tester fills this in. Open / closed / dumpster—see previous slide on dumpsters.**

- **Resolution: The project manager owns this 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.

  - As Designed: The program works as it's supposed to.

  - Need Info: The programmer needs more info from you. She has probably asked a question in the comments.

  - Duplicate: This is just a repeat of another bug report (XREF it on this report.) Duplicates should not close until the duplicated bug closes.

  - Withdrawn: The tester who reported this bug is withdrawing the report.

The Problem Report Form

A typical form includes many of the following fields

- **<u>Resolution version</u>: build identifier**

- **<u>Resolved by</u>: programmer, project manager, tester (if withdrawn by tester), etc.**

- **<u>Resolution tested by</u>: originating tester, or a tester if originator was a non-tester**

- **<u>Change history</u>: datestamped list of all changes to the record, including name and fields changed.**

- **<u>Comments</u>: free-form, arbitrarily long field, typically accepts comments from anyone on the project. Testers programmers, tech support (in some companies) and others have an ongoing discussion of repro conditions, etc., until the bug is resolved. Closing comments (why a deferral is OK, or how it was fixed for example) go here.**

    - This field is especially valuable for recording progress and difficulties with difficult or politically charged bugs.

    - Write carefully. Just like e-mail and usenet postings, it's easy to read a joke or a remark as a flame. Never flame.

The Problem Report Form: Further Reading

- **The best discussion in print of bug reporting and bug tracking system design is probably still the one in my book, <u>Testing Computer Software</u>, chapters 5 & 6. (Not because it's so wonderful but because not enough good stuff has been written since.)**

- **Brian Marick has captured some useful material at his site, <u>www.testingcraft.com</u>. (You should get to know this site, and ideally, contribute to it. This is a collection point for examples.)**

- **Hung Quoc Nguyen (who co-authored TCS 2.0 and is working with us on 3.0) published TrackGear, a web based bug tracking system that has a lot of thought behind it. You can get a 30-day free eval at www.logigear.com.**

- **The Testing Tools FAQ lists other bug tracking software that you can get eval copies. The FAQ is linked from the main comp.software.testing FAQ at <u>http://www.rstcorp.com/resources/hosted.html</u>**

Important Parts of the Report: Problem Summary

This one-line description of the problem is the most important part of the report.

- The project manager will use it in when reviewing the list of bugs that haven't been fixed.

- Executives will read it when reviewing the list of bugs that won't be fixed. They might only spend additional time on bugs with "interesting" summaries.

The ideal summary gives the reader enough information to help her decide whether to ask for more information. It should include:

- A brief description that is specific enough that the reader can visualize the failure.

- A 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.)

The Report:
Can You Reproduce The Problem?

You may not see this on your form, but you should always provide this information.

- Never say it's reproducible unless you have recreated the bug. (Always try to recreate the bug before writing the report.)

- If you've tried and tried but you can't recreate the bug, say "No". Then explain what steps you tried in your attempt to recreate it.

- If the bug appears sporadically and you don't yet know why, say "Sometimes" and explain.

- <u>You may not be able to try to replicate</u> some bugs. Example: customer-reported bugs where the setup is too hard to recreate.

The following policy is not uncommon:

- *If the tester says that a bug is reproducible and the programmer says it is not, then the tester has to recreate it in the presence of the programmer.*

The Report--Description; How to Reproduce It.

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.

- NUMBER THE STEPS. Take it one step at a time.

- If anything interesting happens on the way, describe it. (You are giving people directions to a bug. Especially in long reports, people need landmarks.)

Describe the erroneous behavior and, if necessary, explain what should have happened. (Why is this a bug? Be clear.)

List the environmental variables (config, 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. 45

The Report-- Description;
How to Reproduce It.

It is essential keep the description focused:

The first part of the description should be the shortest step-by-step statement of how to get to the problem.

Add "Notes" after the description if you have them. Typical notes include:

- Comment that the bug won't show up if you do step X between step Y and step Z.

- Comment explaining your reasoning for running this test.

- Comment explaining why you think this is an interesting bug.

- Comment describing other variants of the bug.

Keeping the Report Simple

If you see two failures, write two reports.

Combining failures on one report creates problems:

- The summary description is typically vague. You say words like "fails" or "doesn't work" instead of describing the failure more vividly. This weakens the impact of the summary.

- The detailed report is typically lengthened. It's common to see bug reports that read like something written by an inept lawyer. Do this unless that happens in which case don't do this unless the first thing and then the testcase of the second part and sometimes you see this but if not then that.

- Even if the detailed report is rationally organized, it is longer (there are two failures and two sets of conditions, even if they are related) and therefore more intimidating.

- You'll often see one bug get fixed but not the other.

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

Keeping it Simple:
Eliminate Unnecessary Steps (1)

Sometimes it's not immediately obvious what steps can be dropped from a long sequence of steps in a bug.

- *Look for critical steps -- Sometimes the first symptoms of an error are subtle.*

You have a list of the steps you took to show the error. You're now trying to shorten the list. Look carefully for any hint of an error as you take each step -- A few things to look for:

- Error messages (you got a message 10 minutes ago. The program didn't fully recover from the error, and the problem you see now is caused by that poor recovery.)

- Delays or unexpectedly fast responses.

- Display oddities, such as a flash, a repainted screen, a 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 character or graphic that contained them was erased or moved).

Keeping it Simple:
Eliminate Unnecessary Steps (2)

- Sometimes the first indicator that 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 when nothing is being sent to it (or a light that is off when it shouldn't be).

- Debug messages—turn on the debug monitor on your system (if you have one) and see if/when a message is sent to it.

If you've found what looks like a critical step, try to eliminate almost everything else from the bug report. Go directly from that step to the last one (or few) that shows the bug. If this doesn't work, try taking out individual steps or small groups of steps.

Keep it Simple:
Put Variations After the Main Report

Suppose that the failure looks different under slightly different circumstances. For example, suppose that:

- The timing changes if you do two additional sub-tasks before hitting the final reproduction step
- The failure won't show up or is much less serious if you put something else at a specific place on the screen
- The printer prints different garbage (instead of the garbage you describe) if you make the file a few bytes longer

This is all useful information for the programmer and you should include it. But to make the report clear:

- Start the report with a simple, step-by-step description of the shortest series of steps that you need to produce the failure.
- Identify the failure. (Say whatever you have to say about it, such as what it looks like or what impact it will have.)
- **Then 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.

Overcoming Objections:
Analysis of the Failure

Things that will make programmers resist spending their time on the bug:

- The programmer can't replicate the defect.

- Strange and complex set of steps required to induce the failure.

- Not enough information to know what steps are required, and it will take a lot of work to figure them out.

- The programmer doesn't understand the report.

- ## Unrealistic (e.g. "corner case")

- It's a feature!

Overcoming Objections:
Unrealistic (e.g., Corner Conditions)

Some reports are inevitably dismissed as unrealistic (having no importance in real use).

- If you're dealing with an extreme value, do follow-up testing with less extreme values.
- If you're protesting a bug that has been left unfixed for several versions, realized that it has earned tenure in some people's minds. Perhaps, though, customer complaints about this bug have simply never filtered through to developers.
- If your report of some other type of defect 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 know:
  - -- Technical marketing   -- Technical support
  - -- Human factors   -- Documentation
  - -- Network administrators-- Training
  - -- In-house power users   -- Maybe sales

Overcoming Objections: Analysis of the Failure

Things that will make programmers resist spending their time on the bug:

- The programmer can't replicate the defect.

- Strange and complex set of steps required to induce the failure.

- Not enough information to know what steps are required, and it will take a lot of work to figure them out.

- The programmer doesn't understand the report.

- Unrealistic (e.g. "corner case")

- It's a feature!

   Later in the course, we'll think about this. The usual issues involve the costs of fixing bugs, the company's understanding of the definitions of bugs, and your personal credibility.

Bug Advocacy
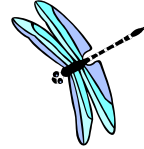
Editing Bug Reports
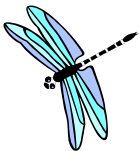
Editing Bug Reports

Some groups have a second tester (usually a senior tester) review reported defects before they go to the programmer. The second tester:

- checks that critical information is present and intelligible
- checks whether she can reproduce the bug
- asks whether the report might be simplified, generalized or strengthened.

If there are problems, she takes the bug back to the original reporter.

- If the reporter was outside the test group, she simply checks basic facts with him.
- If the reporter was a tester, she points out problems with an objective of furthering the tester's training.

Editing Bug Reports

This tester might review:

- all defects

- all defects in her area

- all of her buddy's defects.

In designing a system like this, beware of overburdening the reviewing testers. The reviewer will often go through a learning curve (learning about parts of the system or types of tests that she hasn't studied before). This takes time. Additionally, you have to decide whether the reviewer is doing an actual reproduction of the test or thinking about the plausibility and understandability of the report when she reads it.

Editing Bugs--Practice at Home

Go through your bug database and find some bugs that look interesting

- Do an initial review of them

- Replicate them

- Revise the descriptions to make them clearer and more useful.

Assignment:

- Give two improved bugs to a co-worker

- Review two improved bugs from a co-worker

- Compare notes

(Note: When I teach this course to undergraduates, I require them to successfully edit bugs before they can write any. It is effective training.)

Editing Bugs
Assignment Procedure

First times: The tester gives you the bug report before entering it into the bug tracking system.

- The reporter should give you a hard copy of the proposed bug report or a file in a format you can read. If you can't read the reporter's file format, the reporter has to give you the bug in some other format. This is the reporter's responsibility, not yours.

- Read over the report. If you can't understand it or if there are obvious problems, note those problems and return it to the reporter. If there are significant problems when you try to read the report, don't spend any time trying to replicate it. Just give it back and deal with it again later, when it has been fixed.

- If the report is OK to read (not perfect, but OK), make some comments (maybe on a printout, maybe in the text file that the reporter gave you) and then try to replicate the bug. Make comments as appropriate. Then hand the commented report back to the reporter. The reporter can review your comments, decide what to change, and then either:
  - » Submit the bug directly into the bug tracking system, or
  - » Give the bug back to you for a second review.

- You are only obligated to review a bug once. If you review the bug and bounce it because it is unintelligible, you don't have to accept it back for replication. If you replicated it and gave feedback, you don't have to review the improved version.

- **If the reporter is submitting a bug to you that was previously reviewed by someone else, she MUST give you a copy of the report that she gave to that other person and their comments, along with the new improved report.**

Editing Bugs
Assignment Procedure

Later times:

If the tester gives you the bug report before entering it into the bug tracking system.

- Same procedure as before

If the tester gives you the report AFTER entering it into the bug tracking system

- Review the report for clarity and tone (see "first impressions", next slide) and send comments back to the reporter by email
- Attempt to replicate the bug and send comments back to the reporter by email on the replication steps, your overall impressions, and any follow-up tests you recommend
- You may edit the bug report yourself, but ONLY in the following ways.
  - » Add a comment indicating that you successfully replicated the bug on XXX configuration in YYY build. (This is only valuable if the configuration or build is different from the reporter's.)
  - » Add a comment describing a simpler set of replication steps. Make sure these are clear and accurate.
  - » Add a comment describing why this bug would be important to customers (this is only needed if the bug looks minor or like it won't be fixed. It is only useful if you clearly know what you are talking about, your tone is respectful).
  - » Your comments should NEVER appear critical or disrespectful of the original report or of the person who wrote it. You are adding information, not criticizing what was there.
- If you edit the report in the database, **never change what the reporter has actually written**. You are not changing his work, you are adding comments to it at the end of the report
- Your comments should have your name and the comment date, usually at the start of the comment, for example: "(Cem Kaner, 12/14/01) Here is an alternative set of replication steps:")
- Send the reporter an email, telling her that you have reviewed the report and made changes.

Editing Bugs—A Checklist

The bug editor should check the bug report for the following characteristics:

1. First impressions—when you first read the report:
   - Is the summary short (about 50-70 characters) and descriptive? (see the slide: *Important Parts of the Report: Problem Summaries*)
   - Can you understand the report? As you read the description, do you understand what the reporter did? Can you envision what the program did in response? Do you understand what the failure was?
   - Is it obvious where to start (what state to bring the program to, to replicate the bug)?
   - Is it obvious what files to use (if any)? Is it obvious what you would type?
   - Is the replication sequence provided as a numbered set of steps, which tell you exactly what to do and, when useful, what you will see?
   - Does the report include unnecessary information, personal opinions or anecdotes that seem out of place?
   - Is the tone of the report insulting? Are any words in the report potentially insulting?
   - Does the report seem too long? Too short? Does it seem to have a lot of unnecessary steps? (This is your first impression—you might be mistaken. After all, you haven't replicated it yet. But does it LOOK like there's a lot of excess in the report?)
   - Does the report seem overly general ("Insert a file and you will see" – what file? What kind of file? Is there an example, like "Insert a file like blah.foo or blah2.fee"?)

Editing Bugs—A Checklist

The bug editor should check the bug report for the following characteristics:

2. When you replicate the report:
    - Can you replicate the bug?
    - Did you need additional additional?
    - Did you get lost or wonder whether you had done a step correctly? Would additional feedback (like, "the program will respond like this...") have helped?
    - Did you have to guess about what to do next?
    - Did you have to change your configuration or environment in any way that wasn't specified in the report?
    - Did some steps appear unnecessary? Were they unnecessary?
    - Did the description accurately describe the failure?
    - Did the summary accurate describe the failure?
    - Does the description include non-factual information (such as the tester's guesses about the underlying fault) and if so, does this information seem credible and useful or not?
    - Does the description include

Editing Bugs—A Checklist

The bug editor should check the bug report for the following characteristics:

3. Closing impressions:

- Does the description include non-factual information (such as the tester's guesses about the underlying fault) and if so, does this information seem credible and useful or not? (The report need not include information like this. But it should not include non-credible or non-useful speculation.)

- Does the description include statements about why this bug would be important to the customer or to someone else? (The report need not include such information, but if it does, it should be credible, accurate, and useful.)

Editing Bugs—A Checklist

The bug editor should check the bug report for the following characteristics:

4. Follow-up tests:

    - Are there follow-up tests that you would run on this report if you had the time? (*Refer to the slides on follow-up testing)?*
    - What would you hope to learn from these tests?
    - How important would these tests be?
    - You will probably NOT have time to run follow-up tests yourself, or if you run any, you will not / should not take the time to run more than 1 or 3 such tests.
    - Are some tests so obvious that you feel the reporter should run them before resubmitting the bug? Can you briefly describe them to the reporter?
    - Some obvious style issues that call for follow-up tests—if the report describes a corner case without apparently having checked non-extreme values. Or the report relies on other specific values, with no indication about whether the program just fails on those or on anything in the same class (what is the class?) Or the report is so general that you doubt that it is accurate ("Insert any file at this point" – *really? Any file? Any type of file? Any size?* Maybe this is accurate, but are there examples or other reasons for you to believe this generalization is credible?)

Editing Bugs—Grading Issues

After you have finished editing a bug, send me a copy of the report you received and the comments you sent or the report as you revised it. IF THE REPORTER HAS ENTERED THE BUG INTO THE DATABASE, PLEASE TELL ME WHERE TO FIND THE FINAL BUG REPORT. (What file, what bug number or what to search for.)

You are welcome to send me additional comments that describe how easy or hard it was to work with the reporter or how hard (or not hard) the reporter tried to improve his bug report. YOU ARE NOT REQUIRED TO SEND SUCH COMMENTS. DON'T SEND ANYTHING TO ME THAT YOU DON'T WANT THE REPORTER TO SEE. I MAY SHOW THE REPORTER ANY COMMENTS THAT I RECEIVE.

- I will evaluate whether the final report is worth credits to the tester
- I will evaluate whether your edits are worth credits to you.

Editing Bugs—Notes to the Bug Reporter

If you are the tester who wrote the bug report (the reporter), send me a note when you have written a report, entered it into the database and consider it finished. Here are some ground rules:

- Your first two bugs must go through an editing pass by a bug replicator before you can put them into the database. After that, it is up to you whether to submit bugs to replicators or just enter them into the database directly.

- It is up to you to make changes or not make changes to your report, based on the editor's comments. This is your bug report, it carries your signature, write it your way.

- Never, ever write a report under someone else's name.

- If you submitted a report to one editor and their edit was not helpful, you can submit it to another editor. However, when you do this, you must give the second editor a copy of the report that you gave to the first editor and a copy of the first editor's comments.

- When you are satisfied with your bug report, enter it into the bug tracking system in the form you consider final, and send me a note telling me where to find the report. I'll look it over and assign credit (or not).

Black Box Testing

Bug Reporting Exercises

Bug Reporting Exercise 1 (1)

Create a sample database of cheques. Enter many new cheques.

Bug Reporting Exercise 1 (2)

Quicken Find

Find: testing

Search: All Fields

Match if: Contains

☑ Search Backwards

▶ Fi**nd**

▦ Find All

◪ Close

? Help
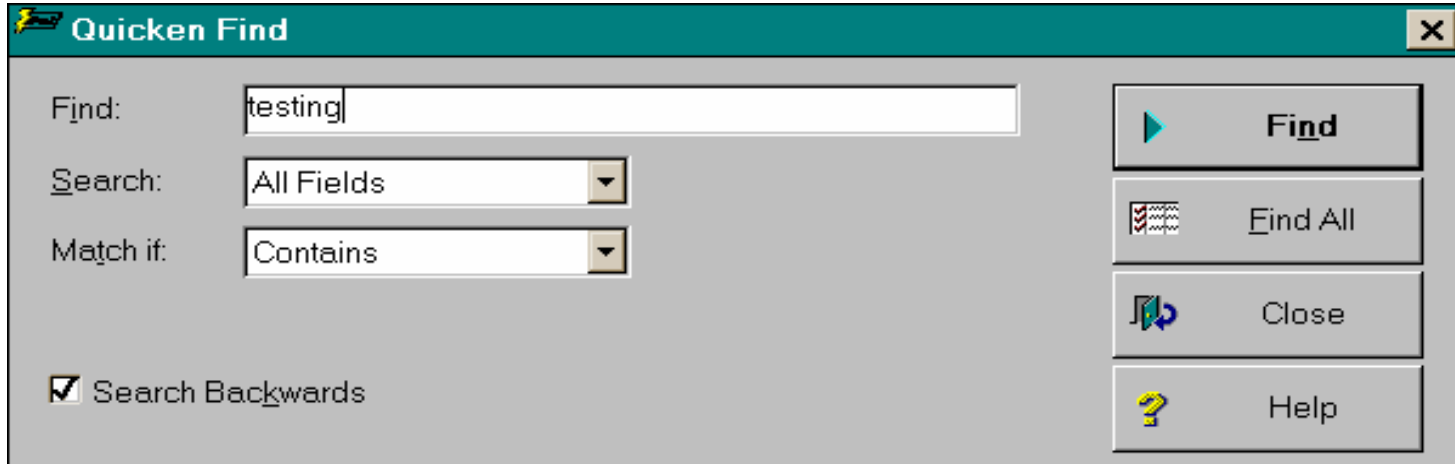
Now search the cheques to find one. Here, I searched for the word "testing". The program searches backwards, from the currently selected cheque to the start of the register. It doesn't find any instances of "testing" so it asks whether it should keep searching from the end of the register backwards.

Quicken for Windows

? Continue search from end of register?

✔ Yes   ⊘ No
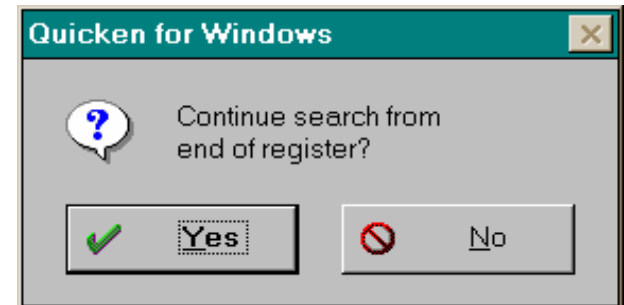
Bug Reporting Exercise 1 (3)
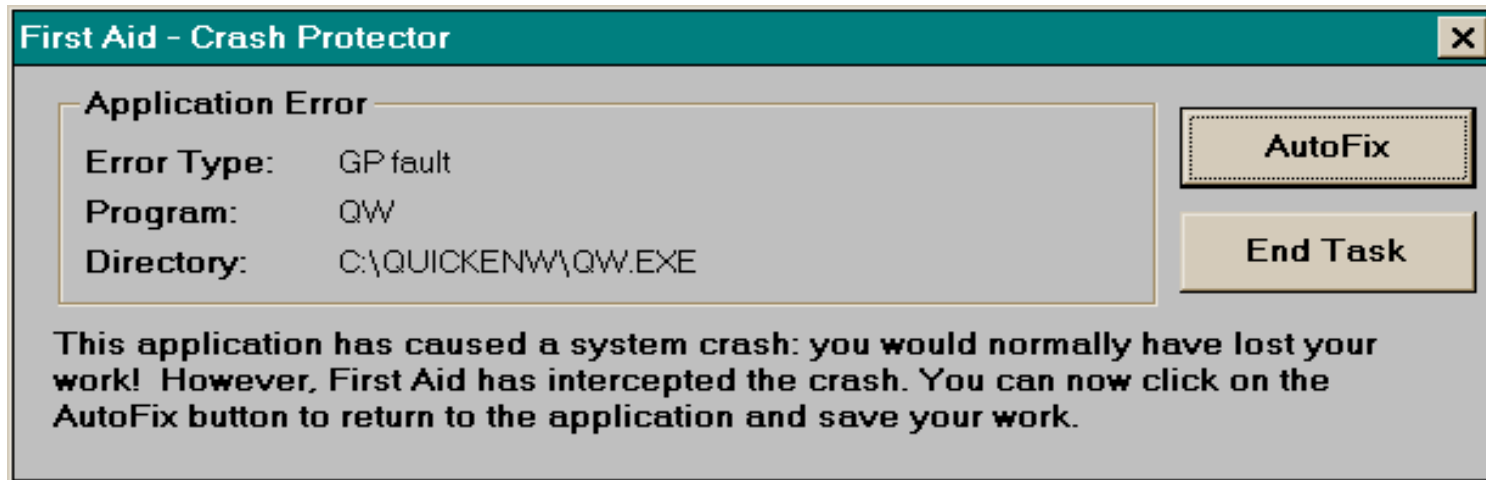
Kaboom! A General Protection Fault!

- The "First Aid" application tries to protect the customer from losing data when there is a GP fault. It's always *possible* that the crash was *caused* by an interaction between Quicken and First Aid, so try the test again after turning off First Aid.
- When I re-ran the test, Quicken crashed again, with a Win 95 system window that identifies a GP Fault. (These are harder to screen shoot, so it's not here.) Therefore the bug was not due to First Aid.

Bug Reporting Exercise 1 (4)

Quicken Find

Find: testing

Search: All Fields

Match if: Contains

☑ Search Backwards

▶ Find

Find All

Close

? Help
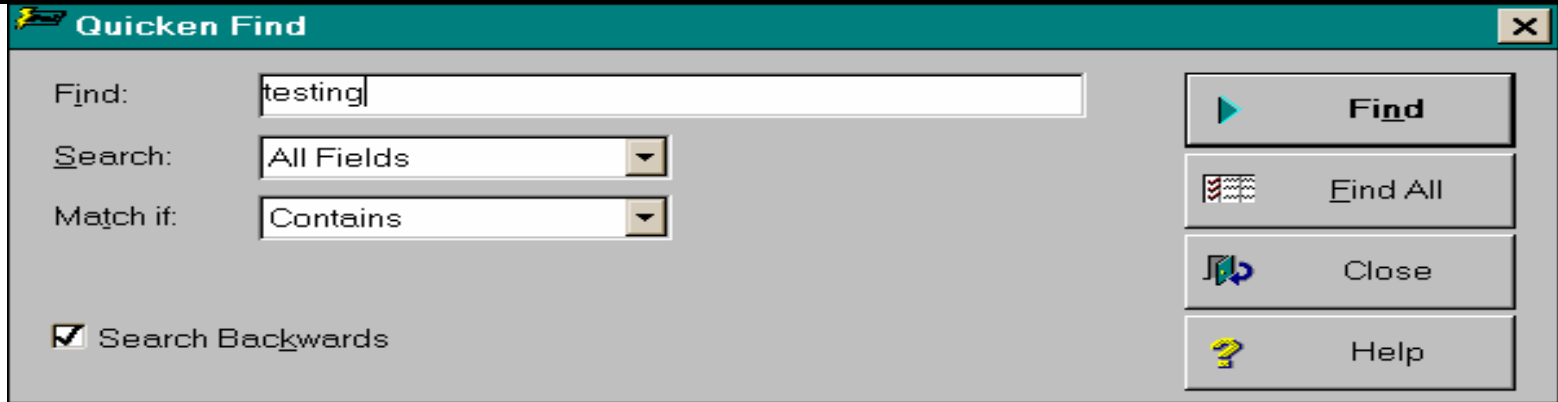
When analyzing a bug, it's wise to try to recreate it on another computer. I did that. This time, the search didn't crash. The crashing computer is a Pentium with 32 megs RAM, a Logitech trackball, the MS keyboard, a 1.6 gig hard drive, no disk compression, a 4 meg high res MPEG video card and a big monitor. The other is an 8 meg 486 with an MS Mouse, an old standard keyboard, a 540 meg hard drive (compressed) and basic SVGA video.

Quicken for Windows

? Continue search from end of register?

✔ Yes   🚫 No

Quicken for Windows

ℹ No matching transactions were found.

✔ OK
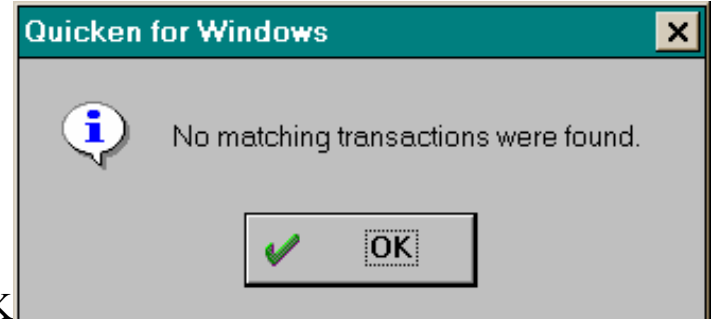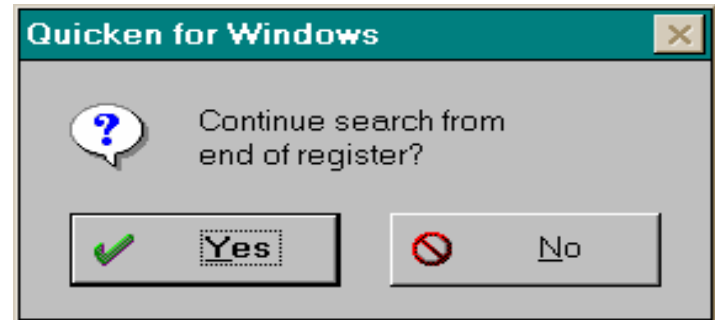
Bug Reporting Exercise 1 (5)

Because this is a crash, you decide to get it into the tracking system right away. You'll do more troubleshooting later. So here is your assignment.

1   Write these two sections of the bug report:
   » Problem Summary
   » Problem Description

2   What other tests should you run? Why? Write down your list.

3   Meet with your group to read each other's reports.
   » How good is the summary?
   » How clear is the description?
   » How complete is the description?
   » How accurate is the description?
   » How promising is your list of ideas?

Notes

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Notes

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Notes

Notes on Exercise

I do some analysis before writing. Here's a structure for making your notes:

OBSERVED FAILURES	CONDITIONS
• *General protection fault*	• *search for non-existent text* • *search backwards* • *Yes to query, search from end of register*
OTHER CONDITIONS (maybe irrelevant) • *Configurations (list them all)*	NOTES

Notes on Exercise

<u>MY SUMMARY</u>

GPF on search for non-existent text. (Configuration dependent.)

<u>MY PROBLEM DESCRIPTION</u>

1. *Start the program*
2. *Open a database (I used the TESTING file)*
3. *Search (backwards) for a string that doesn't appear in the database*
4. *When the program asks whether to search from end of register, click YES*
5. *Result = GPF*

<u>NOTES</u>: This bug is configuration dependent. The two machines involved are the two at my desk, if you need to replicate while I'm gone. I'll do further analysis later, but I put this into the database now in order to give you an early warning of a serious bug. The configurations of the two machines are:

<u>Replicates</u>	*<u>Fails to Replicate</u>*
Pentium	*486*
etc	*etc*

Notes

Bug Reporting Exercise 2 (1)

The following group of slides are from Windows Paint 95. Please don't spend your time replicating the steps or the bug. (You're welcome to do so if you are curious, but it is not necessary for analysis of this exercise.)

Treat the steps that follow as fully reproducible. If you go back to ANY step, you can reproduce it.

In case you aren't familiar with paint programs, the key idea is that you lay down dots. For example, when you draw a circle, the result is a set of dots, not an object. If you were using a draw program, you could draw the circle and then later select the circle, move it, cut it, etc. In a paint program, you cannot select the circle once you've drawn it. You can select an area that includes the dots that make up the circle, but that area is simply a bitmap and none of the dots in it have any relationship to any of the others.

Here's the opening screen. The background is white. The first thing that we'll do is select the Paint Can

We'll use this to lay down a layer of grey paint on top of the background. Then, when we cut or move an area, we'll see the white background behind what was moved.
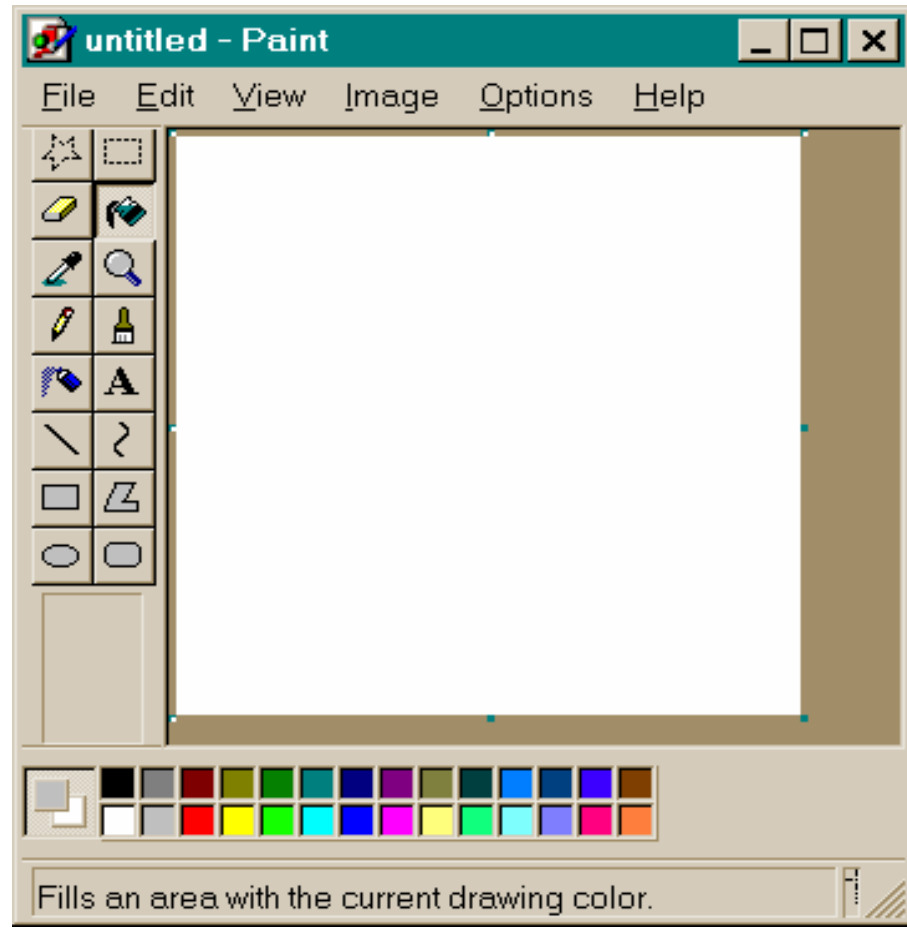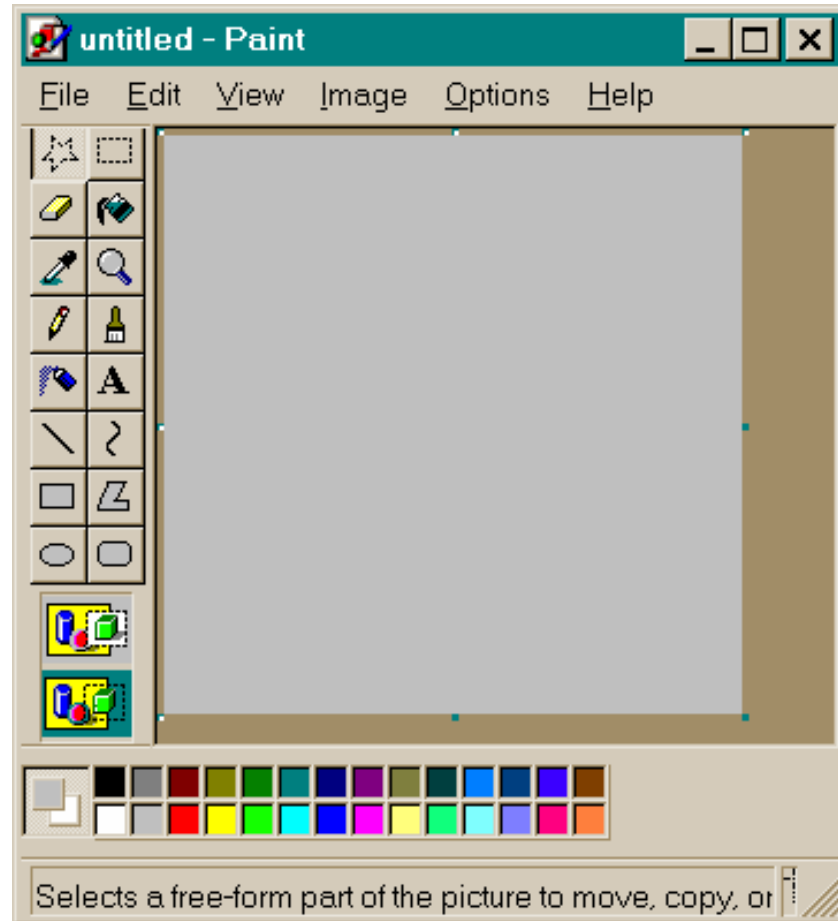
Bug Exercise 2 Continued

**Here's the screen again, but the background has been painted gray.**

The star in the upper left corner is a freehand selection tool. After you click on it, you can trace around any part of the picture. The tracing selects that part of the picture. Then you can cut it, copy it, move it, etc.

Bug Exercise 2 Continued

This shows an area selected with the freehand selection tool. The bottom right corner is selected. (The dashed line surrounds the selected area.)

NOTE: The actual area selected might not be perfectly rectangular. The freehand tool shows a rectangle that is just big enough to enclose the selected area. For our purposes, this is not a bug. This is a design decision by Microsoft.
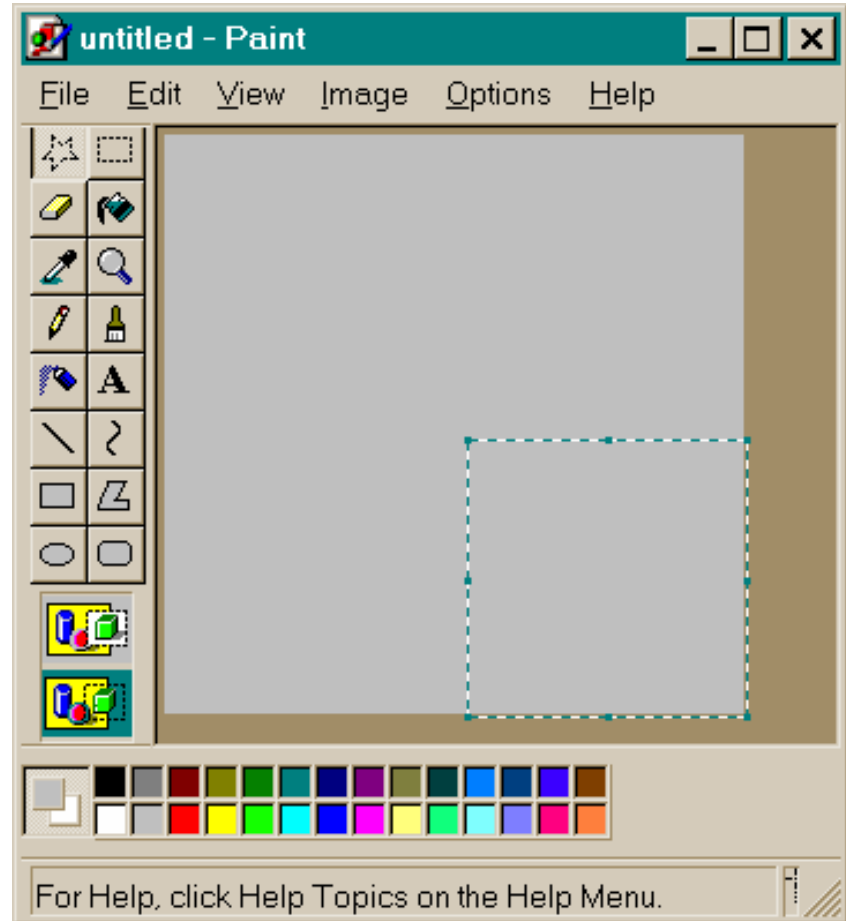
Bug Exercise 2 Continued

Next, we'll draw a circle (so you can see what's selected), then use the freehand select tool to select the area around it.
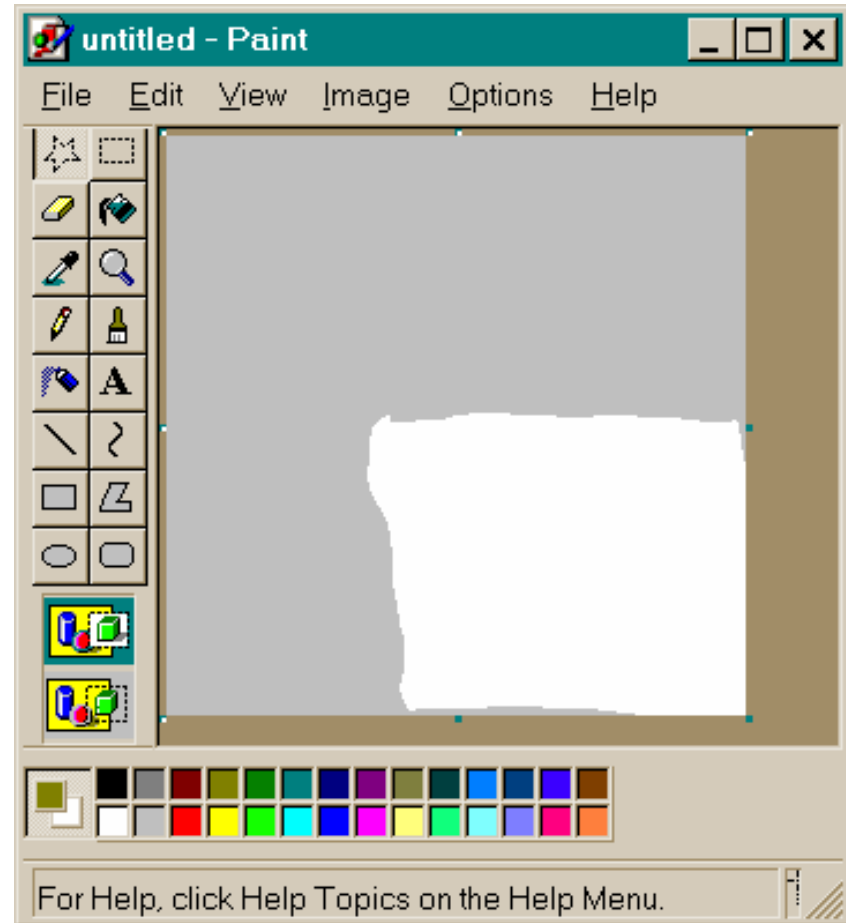
When you use the freehand selection tool, you select an area by moving the mouse. The real area selected is not a perfect rectangle. The rectangle just shows us where the selected area is.
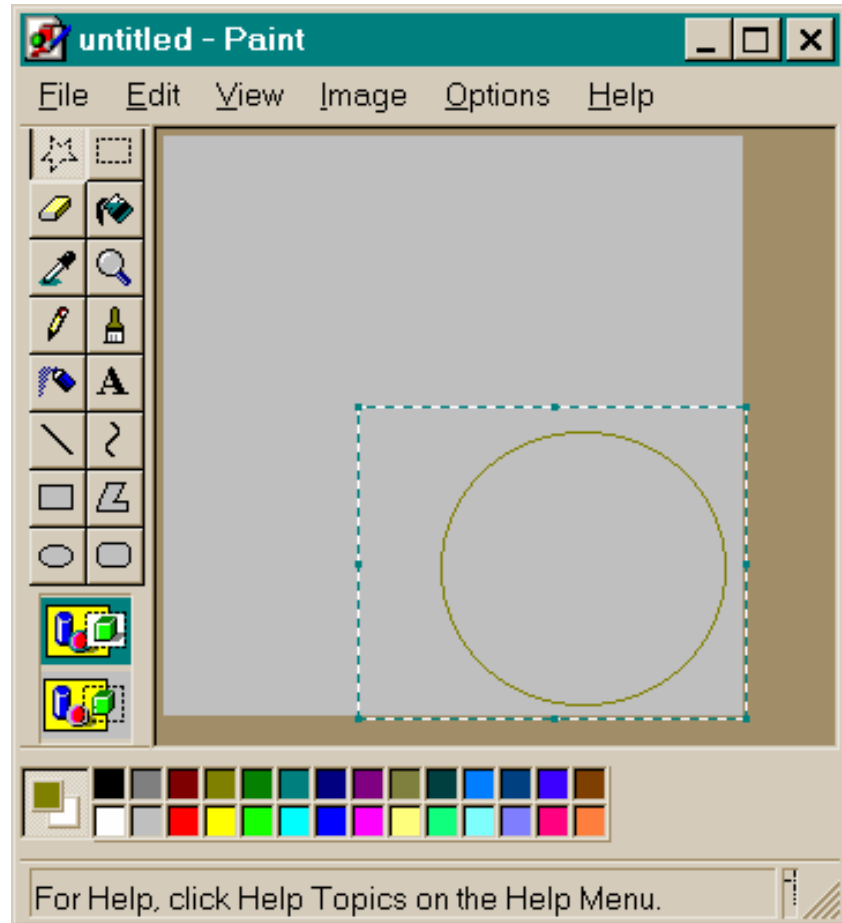
Bug Exercise 2 Continued

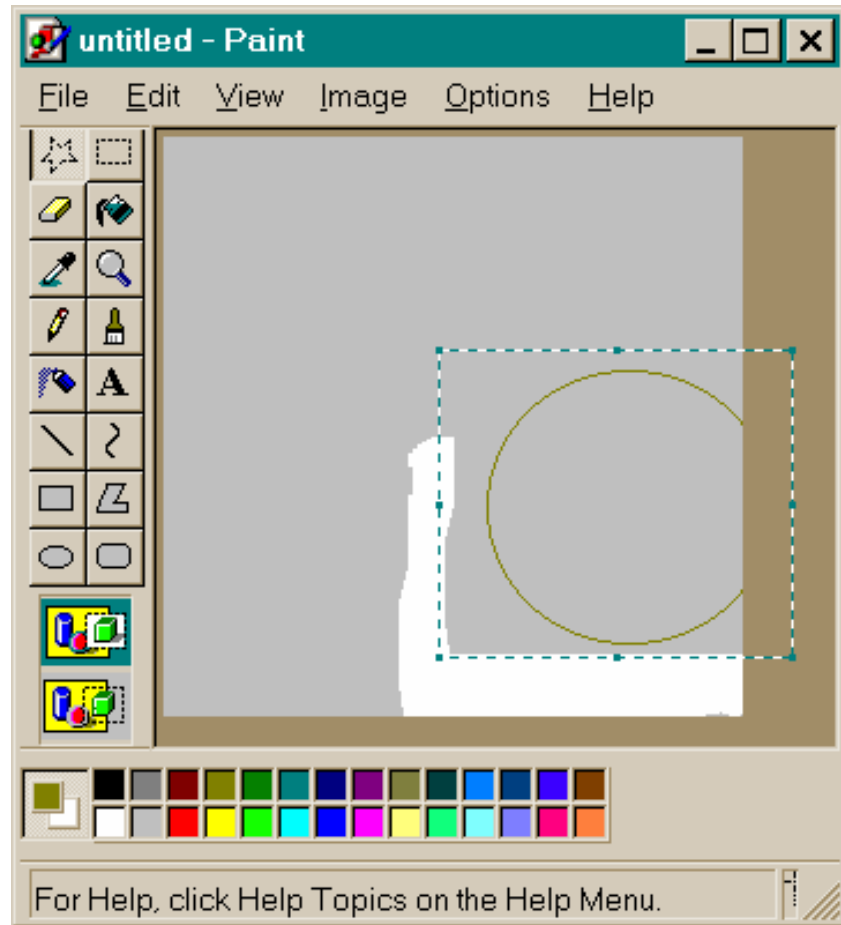**Now we cut the selection. (To do this, press Ctrl-X.)**

The jagged border shows exactly the area that was selected.

Bug Exercise 2 Continued

Next, select the area around the circle and drag it up and to the right.
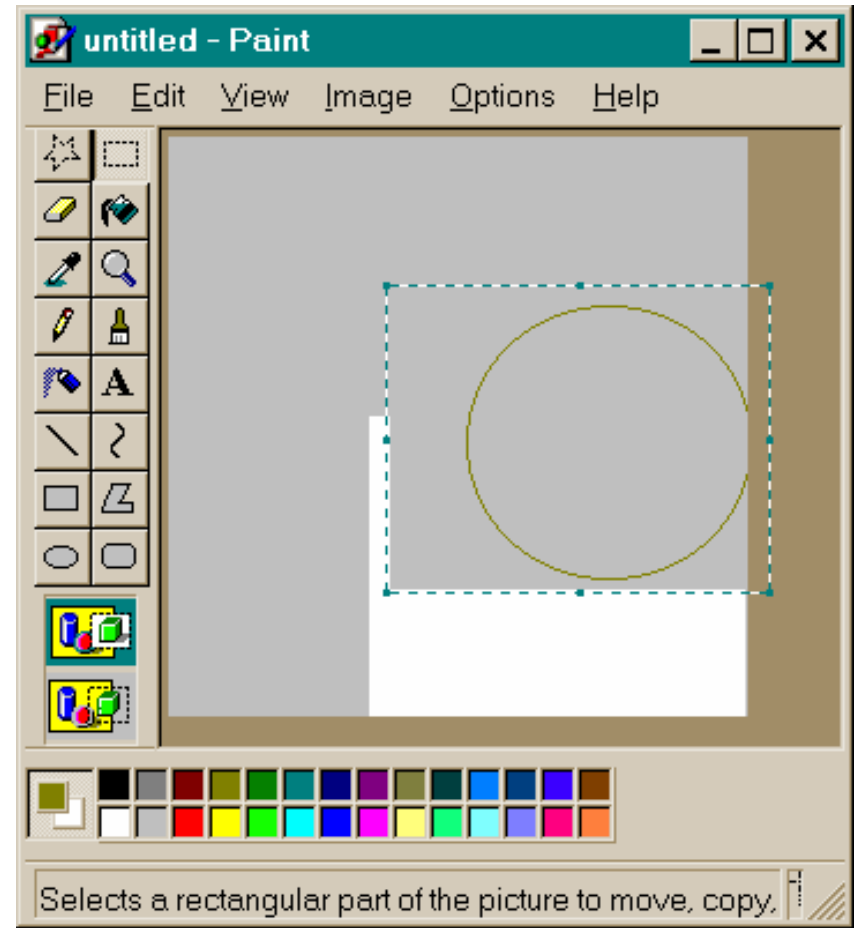
This works.

84

Bug Exercise 2 Continued

This time, we'll try the Rectangular Selection tool.

With this one, if you move the mouse to select an area, the area that is actually selected is the smallest rectangle that encloses the path that your mouse drew.

So, draw a circle, click the Rectangular Selection tool, select the area around the circle and move it up. It works.

85

Bug Exercise 2 Continued

Well, this was just too boring, because everything is working. When you don't find a bug while testing a feature, one tactic is to keep testing the feature but combine it with some other test.

In this case, we'll try Zooming the image. When you zoom 200%, the picture itself doesn't change size, but the display doubles in size. Every dot is displayed as twice as tall and twice as wide.
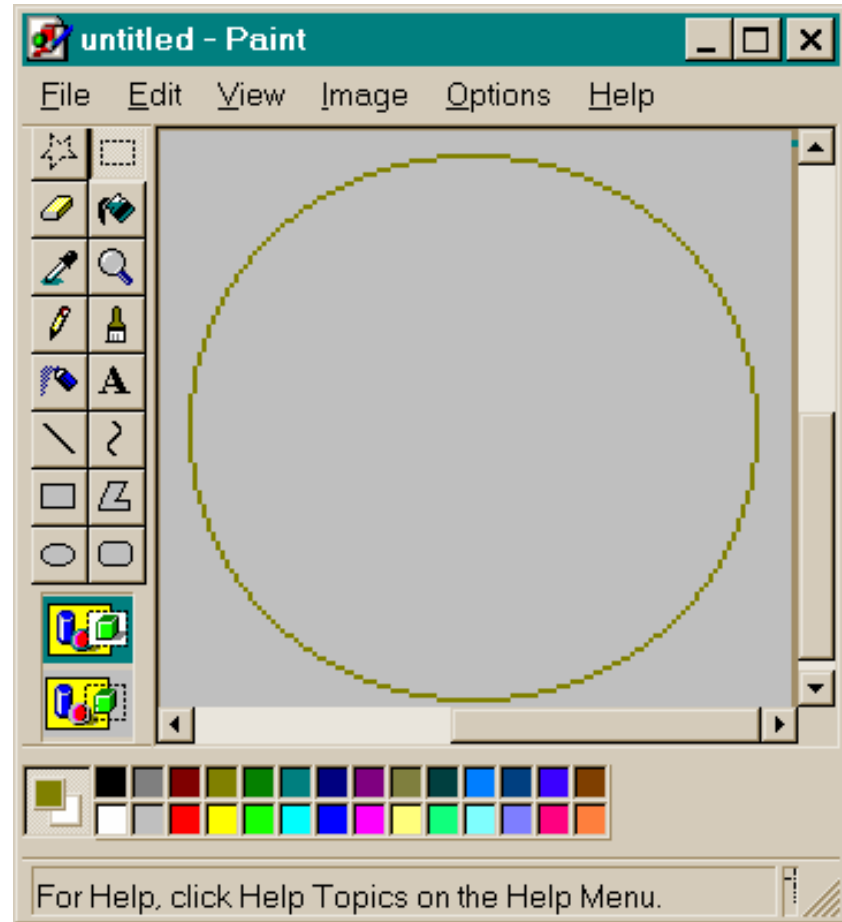
Bug Exercise 2 Continued

Bring up the Custom Zoom dialog, and select 200% zoom, click OK.
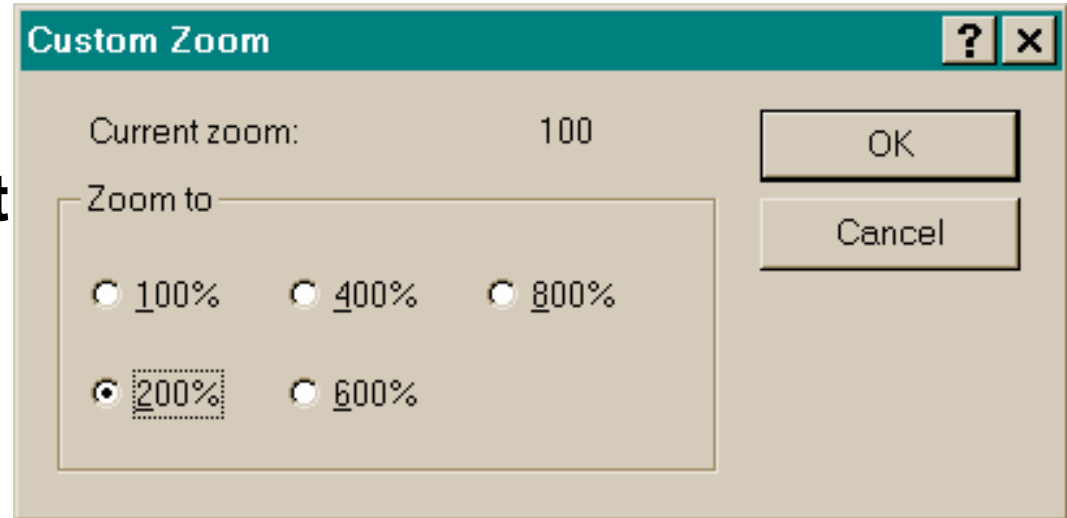
Bug Exercise 2 Continued

It worked. The paint area is displayed twice as tall and twice as wide. We're looking at the bottom right corner. To see the rest, we could move the scroll bars up or left.
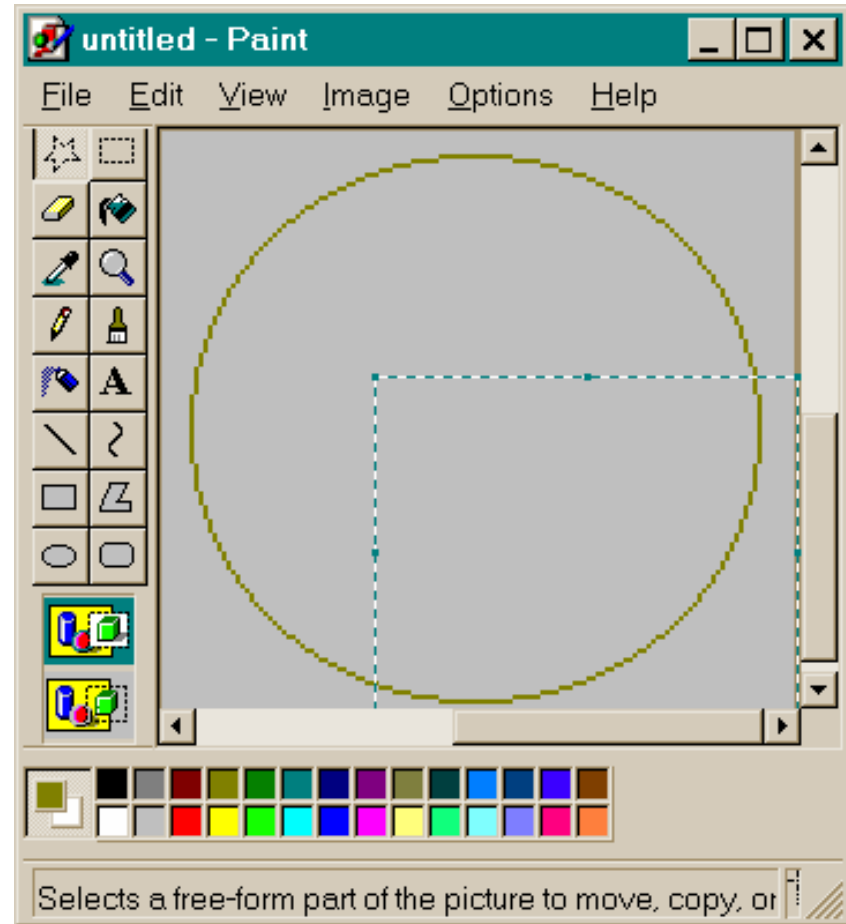
So, we select part of the circle using the freehand selection tool. We'll try the move and cut features.
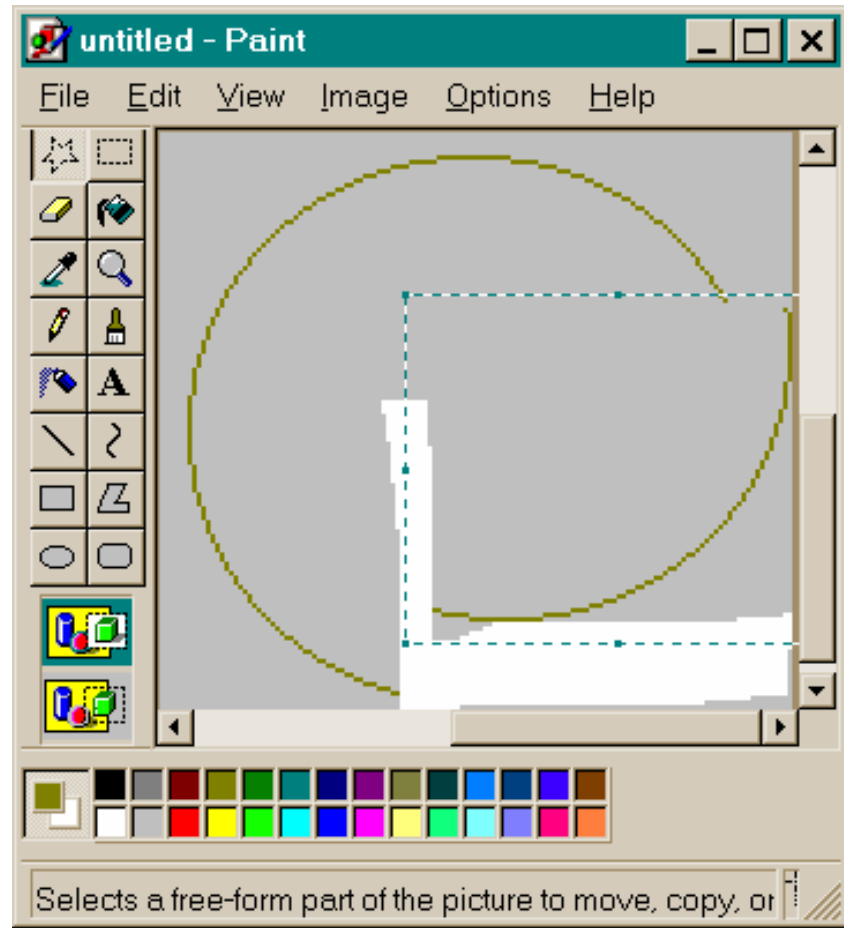
Cutting fails.

When we try to cut the selection, the dashed line disappears, but nothing goes away.

Bug Exercise 2 Continued

Draw the circle, zoom to 200%, select the area.
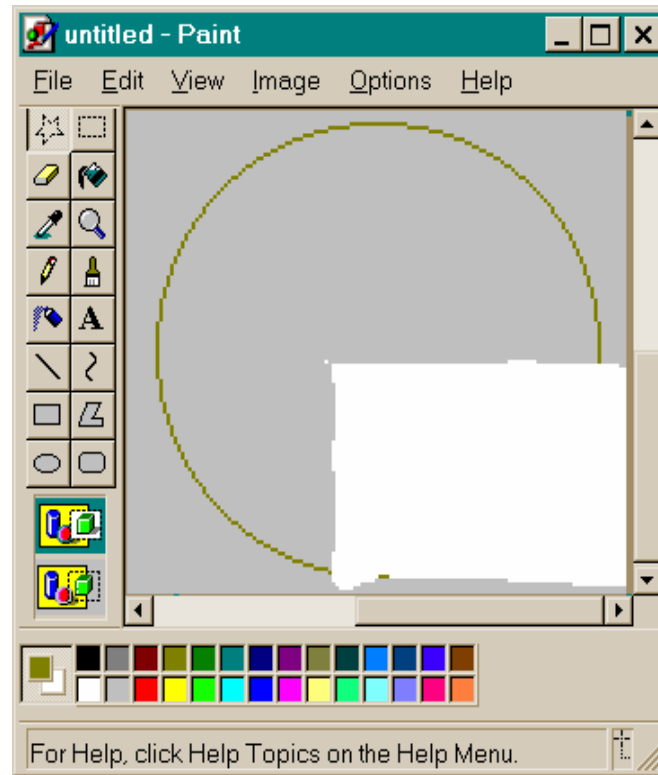
Drag the area up and to the right. It works.

Bug Exercise 2 Continued

Draw the circle, zoom to 200%, select the area.
Now try this. Select the area and move it a bit. THEN
press Ctrl-X to cut. This time, cutting works.
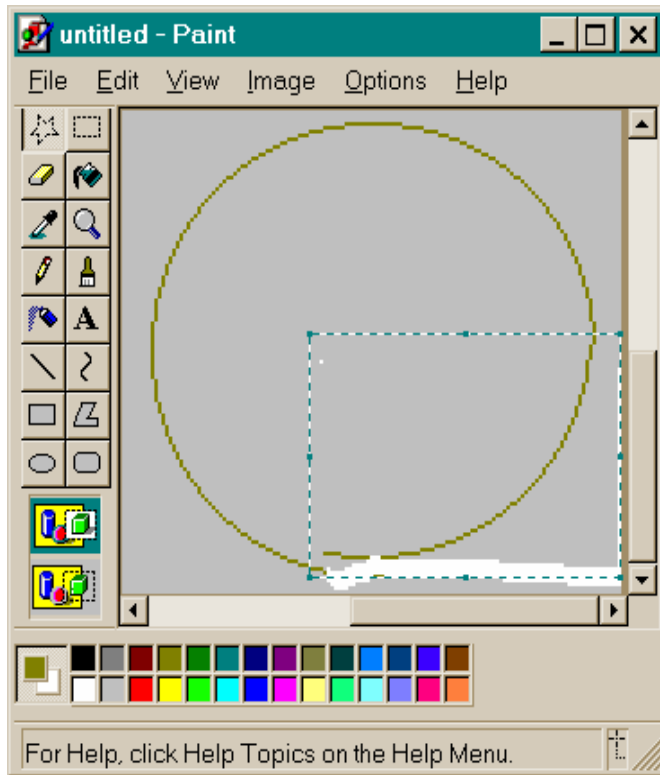
Bug Exercise 2 Continued

Draw the circle, zoom to 200%, and this time, *grow the window* so you can see the whole drawing area.
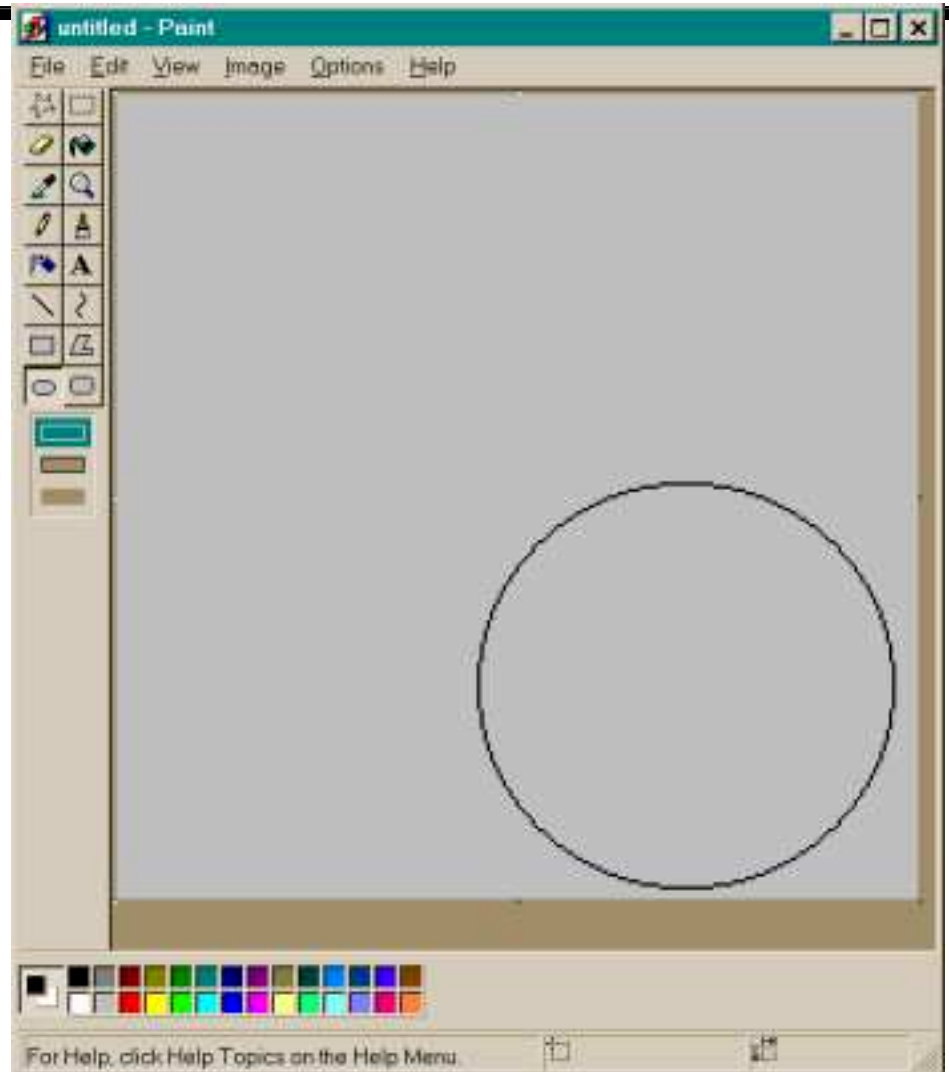
Bug Exercise 2 Continued

Now, select the circle. That seems to work.

93
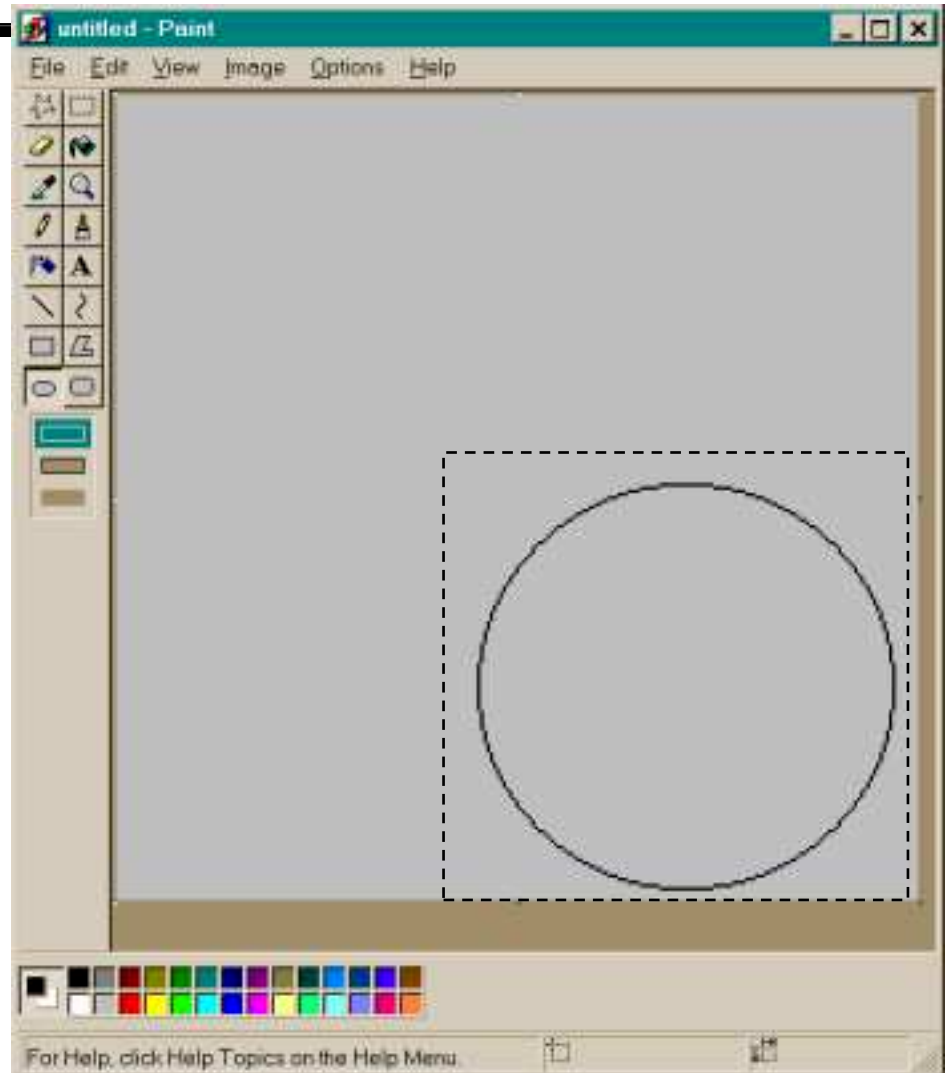
Bug Exercise 2 Continued

But when you press Ctrl-X to cut the circle, the program cuts the wrong area.

94
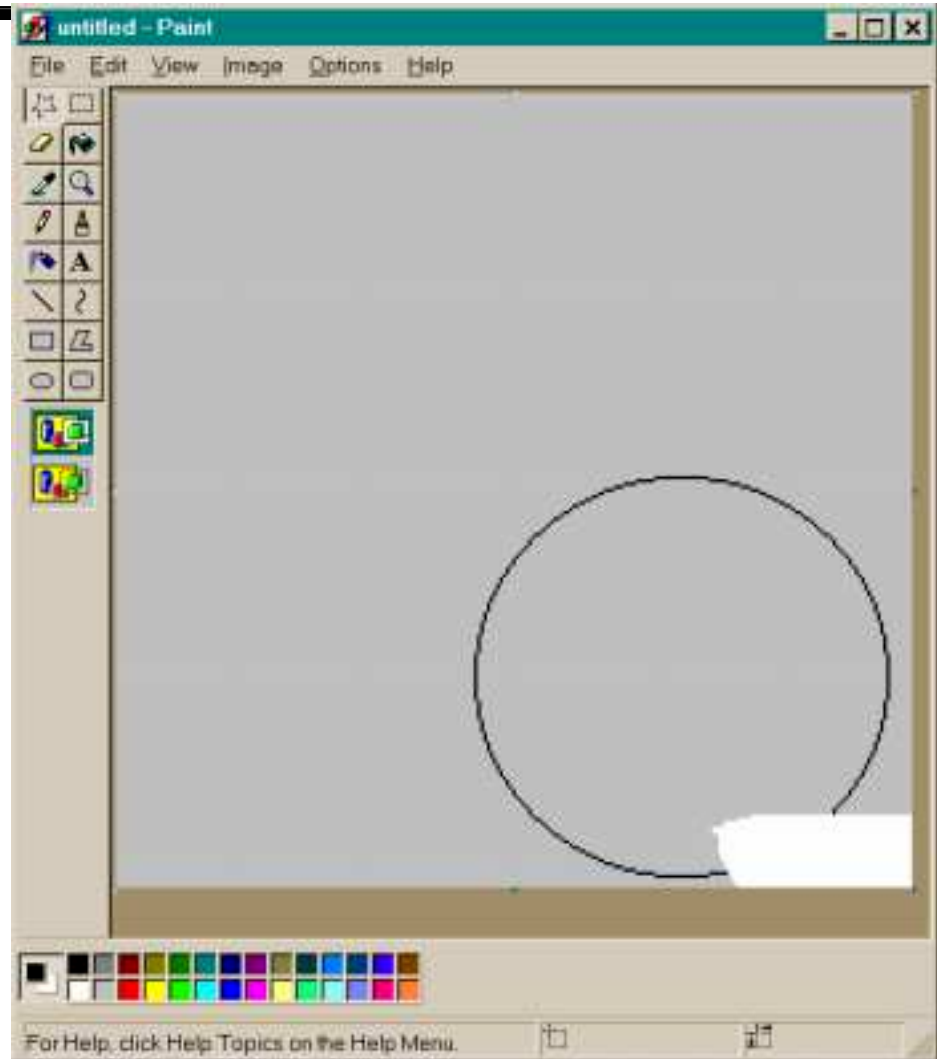
Bug Exercise 2 Continued

Now, write a bug report. I want two sections:
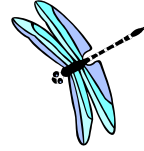
- The Problem summary (or title)

- The Problem Description (how to reproduce the problem)

Additionally, please describe three follow-up tests that you would run with this bug

Notes

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Bug Advocacy

Advocating for
bug fixes
by alerting people
to costs.

Supplementary Reading:

Kaner, *Quality Cost Analysis: Benefits & Risks.*

Copyright (c) 1994-2000 Cem Kaner.
All rights reserved.

Money Talks:
Cost of Finding and Fixing Software Errors
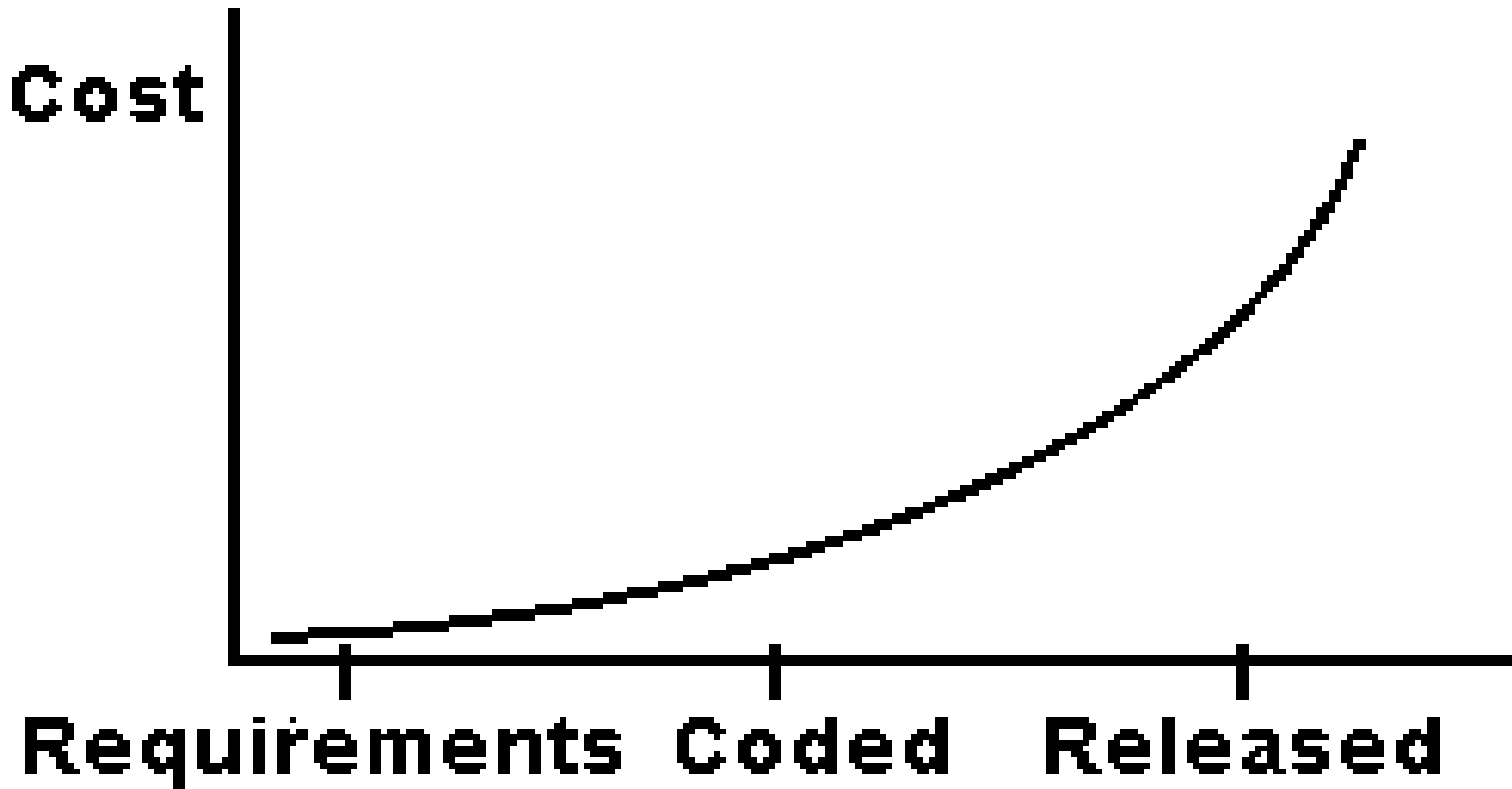
This curve maps the traditionally expected increase of cost as you find and fix errors later and later in development.

Money Talks:
Cost of Finding and Fixing Software Errors

This is the most commonly taught cost curve in software engineering. Usually people describe it from the developers-eye view. That is, the discussion centers around

- how much it costs to find the bug
- how much it costs to fix the bug
- and how much it costs to distribute the bug fix.

But sometimes, it pays to adopt the viewpoints of other stakeholders, who might stand to lose more money than the development and support organizations.

99
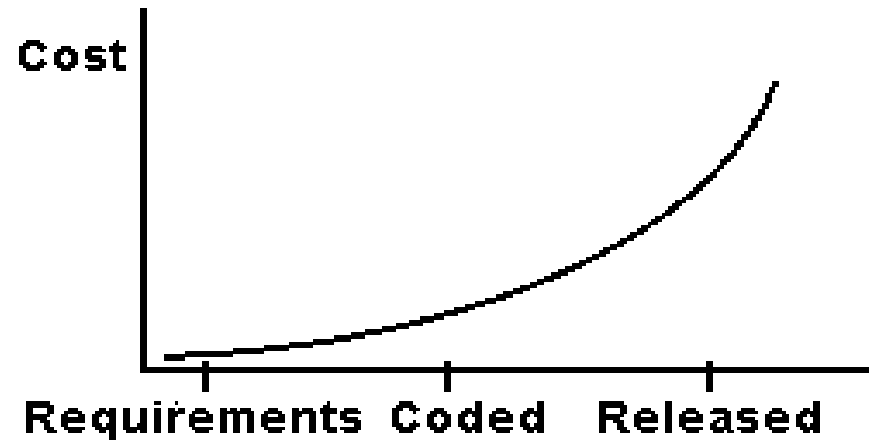
Money Talks:
Cost of Finding and Fixing Software Errors

Costs escalate because more people in and out of the company are affected by bugs, and more severely affected, as the product gets closer to release. We all know the obvious stuff

- if we find bugs in requirements, we can fix them without having to recode anything;
- programmers who find their own bugs can fix them without taking time to file bug reports or explain them to someone else;
- it is hugely expensive to deal with bugs in the field (in customers' hands).

Along with this, there are many effects on other stakeholders in the company. For example, think of the marketing assistant who wastes days trying to create a demo, but can't because of bugs.
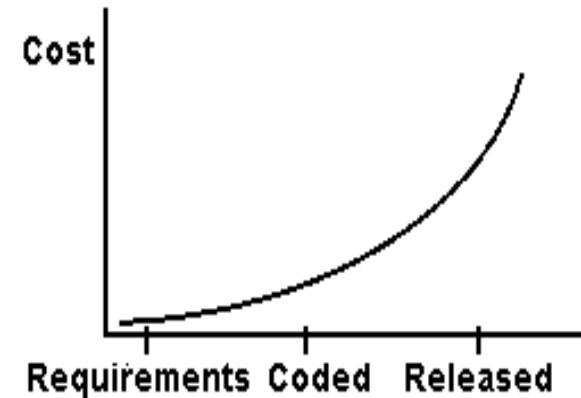
Money Talks:
Cost of Finding and Fixing Software Errors

It is important to recognize that this cost curve is predicated on a family of development practices.

When you see a curve that says,

> "Late changes are expensive"

you can reasonably respond in either of two ways:

- Make fewer late changes.

    - *This is the traditional recommendation*

- **Make it cheaper to make late changes.**

    - *This is a key value of the agile development movement (see Beck's Extreme Programming Explained, or go to www.agilealliance.org)*
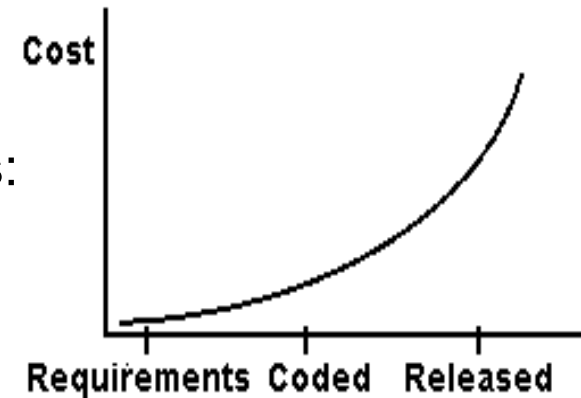
In this testing course, I will push you to find ways to find bugs earlier, but my development philosophy is agile.

Quality Cost Analysis

Quality Cost Measurement is a cost control system used to identify opportunities for reducing the controllable quality-related costs

The *Cost of Quality* is the total amount the company spends to achieve and cope with the quality of its product.

This includes the company's investments in improving quality, and its expenses arising from inadequate quality.

A key goal of the quality engineer is to help the company minimize its cost of quality.

» Refer to the paper, "Quality Cost Analysis: Benefits & Risks."

Quality-Related Costs

Prevention	*Appraisal*
Cost of preventing customer dissatisfaction, including errors or weaknesses in software, design, documentation, and support.	Cost of inspection (testing, reviews, etc.).
Internal Failure	*External Failure*
Cost of dealing with errors discovered during development and testing. Note that the company loses money as a user (who can't make the product work) and as a developer (who has to investigate, and possibly fix and retest it).	Cost of dealing with errors that affect your customers, after the product is released.

Examples of Quality Costs

Prevention	Appraisal
• Staff training • Requirements analysis & early prototyping • Fault-tolerant design • Defensive programming • Usability analysis • Clear specification • Accurate internal documentation • Pre-purchase evaluation of the reliability of development tools	• Design review • Code inspection • Glass box testing • Black box testing • Training testers • Beta testing • Usability testing • Pre-release out-of-box testing by customer service staff
Internal Failure	**External Failure**
• Bug fixes • Regression testing • Wasted in-house user time • Wasted tester time • Wasted writer time • Wasted marketer time • Wasted advertisements • Direct cost of late shipment • Opportunity cost of late shipment	• Lost sales and lost customer goodwill • Technical support calls • Writing answer books (for Support) • Investigating complaints • Supporting multiple versions in the field • Refunds, recalls, warranty, liability costs • Interim bug fix releases • Shipping updated product • PR to soften bad reviews

Customers' Quality Costs

Seller: external costs	Customer: failure costs (seller's externalized costs)
These illustrate costs absorbed by the seller that releases a defective product. • Lost sales and lost customer goodwill • Technical support calls • Writing answer books (for Support) • Investigating complaints • Refunds, recalls, warranty, liability costs • Government investigations • Supporting multiple versions in the field • Interim bug fix releases • Shipping updated product • PR to soften bad reviews • Discounts to resellers	*These illustrate costs absorbed by the customer who buys a defective product.* • Wasted time • Lost data • Lost business • Embarrassment • Frustrated employees quit • Failure during one-time-only tasks, e.g. demos to prospective customers • Cost of replacing product • Reconfiguring the system • Cost of recovery software • Tech support fees • Injury / death

Influencing Others Based on Costs

It's often impossible to fix every bug. But sometimes the development team will choose to not fix a bug based on their assessment of its risks for them, without thinking of the costs to other stakeholders in the company.

- Probable tech support cost.
- Risk to the customer.
- Risk to the customer's data or equipment.
- Visibility in an area of interest to reviewers.
- Extent to which the bug detracts from the use of the program.
- How often will a customer see it?
- How many customers will see it?
- Does it block any testing tasks?
- Degree to which it will block OEM deals or other sales.

To argue against a deferral, ask yourself which stakeholder(s) will pay the cost of keeping this bug. Flag the bug to them.

Notes

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____
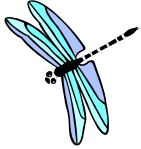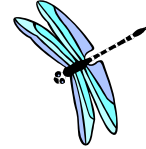
What About the Objection That It Is Not Really A Bug?

- **Really, it's a feature.**
- **Or, at least, it's not a problem for my release so I don't have to fix it.**
- **It won't matter until we ship it to Germany. Let them fix it.**

Supplemental reading: Kaner, What is a Software Defect?

Software Errors: What is Quality?

Here are some of the traditional definitions:

- Fitness for use (Dr. Joseph M. Juran)

- The totality of features and characteristics of a product that bear on its ability to satisfy a given need (ASQ)

- Conformance with requirements (Philip Cosby)

- 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)

Note the absence of "conforms to specifications."

Software Errors:
What is Quality?

Juran distinguishes 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 the customer's equipment

Software Errors: What Should We Report?

I like Gerald Weinberg's definition:

Quality is value to some person

But consider the implication:

- **It's appropriate to report any deviation from high quality as a software error.**

- **Therefore many issues will be reported that will be errors to some and non-errors to others.**

___

Glen Myers' definition:

- A software error is present when the program does not do what its user reasonably expects it to do.

Quality is Multidimensional

Programming	User Interface Design	Marketing
Glass Box Testing		**Black Box Testing**
Multimedia Production	Project Manager	Writing
Content Development	Manufacturing	Customer Service
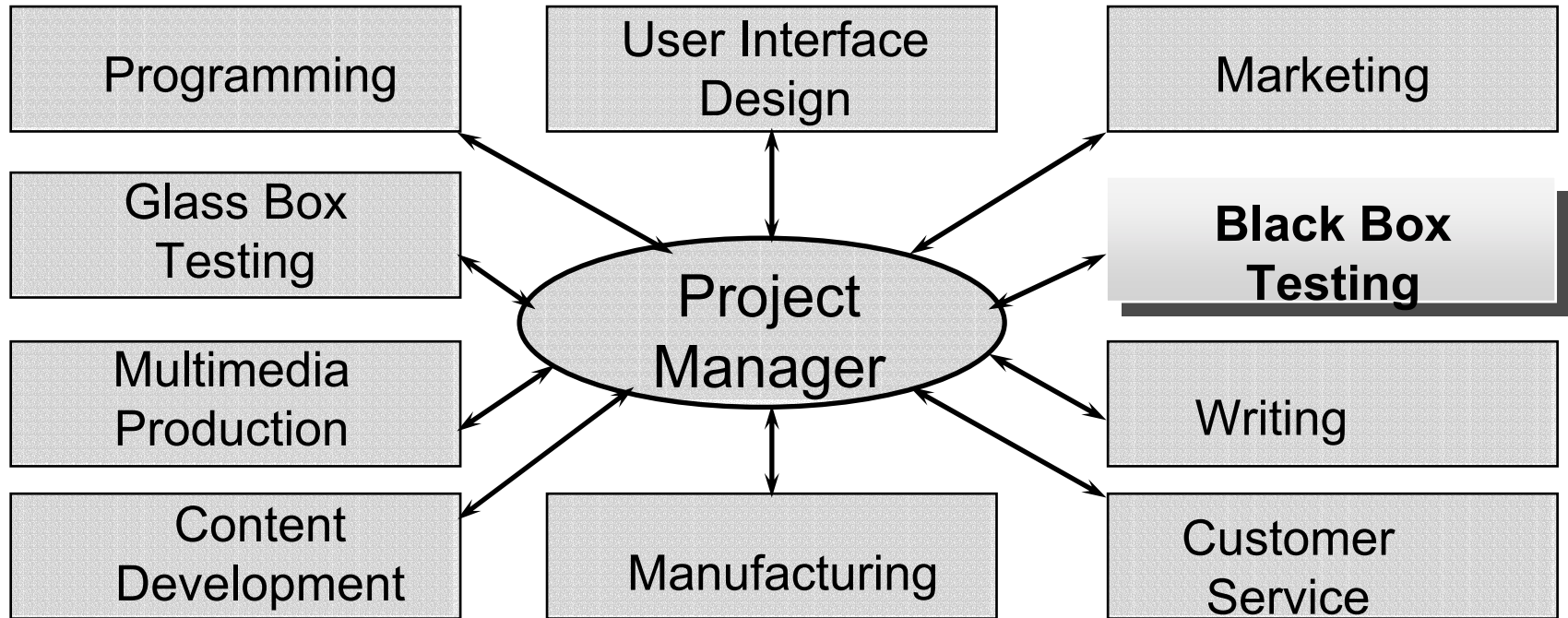
When you sit in a project team meeting, discussing a bug, a new feature, or some other issue in the project, you must understand that each person in the room has a different vision of what a "quality" product would be. Fixing bugs is just one issue.
The next slide gives some examples.

112

Quality is Multidimensional:
Different People, Different Visions

Localization Manager: *A good product is easy to translate and to modify to make it suitable for another country and culture. Few expereienced localization managers would consider acceptable a product that must be recompiled or relinked to be localized.*

Tech Writers: *A high quality program is easily explainable. Aspects of the design that are confusing, unnecessarily inconsistent, or hard to describe are marks of bad quality.*

Marketing: *Customer satisfiers are the things that drive people to buy the product and to tell their friends about it. A Marketing Manager who is trying to add new features to the product generally believes that he is trying to improve the product.*

Customer Service: *Good products are supportable. They have been designed to help people solve their own problems or to get help quickly.*

Programmers: *Great code is maintainable, well documented, easy to understand, well organized, fast and compact.*

113

Software Errors:
What Kind of Error?

You will report all of these types of problems, but it's valuable to keep straight in your mind, and on the bug report, which type you're reporting.

- Coding Error: *The program doesn't do what the programmer would expect it to do.*

- Design Issue: *It's doing what the programmer intended, but a reasonable customer would be confused or unhappy with it.*

- Requirements Issue: *The program is well designed and well implemented, but it won't meet one of the customer's requirements.*

- 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: *Sometimes the spec is right; sometimes the code is right and the spec should be changed.*

Software Errors:
Why are there Errors?

New testers often conclude that the programmers on their project are incompetent or unprofessional.

- This is counterproductive. It leads to infighting instead of communication, and it leads to squabbling over bugs instead of research and bug fixing.

- And as we saw when we discussed private bug rates, programmers actually find and fix the large majority of their own bugs.

- Bugs come into the code for many reasons. It's worth considering some common systematic (as distinct from poor individual performance) factors. You will learn to vary your strategic approaches as you learn your companies' weaknesses.

Software Errors:
Why are there Errors?

Bugs come into the code for many reasons:

- The major cause of error is that programmers deal with tasks that aren't fully understood or fully defined. This is said in many different ways. For example:

  » Tom Gilb and Dick Bender quote industry-summary statistics that 80% of the errors, or 80% of the effort required to fix the errors, are caused by bad requirements;

  » Roger Sherman recently summarized research at Microsoft that the most common underlying issue in bug reports involved a need for new code.

  If you graduated from a Computer Science program, how much training did you have in task analysis? Requirements definition? Usability analysis? Negotiation and clear communication of negotiated agreements? Not much? Hmmmm . . . .

116

Software Errors:
Why are there Errors?

- Some companies drive their programmers too hard. They don't have enough time to design, bulletproof, or test their code. Another Sherman quote: "Bad schedules are responsible for most quality problems."

- Late design changes result in last minute code changes, which are likely to have errors.

- Some third-party components introduce bugs. Your program might rely on a large suite of small components that display a specific type of object, filter data in a special way, drive a specific printer, etc. Many of these tools, bought from tool vendors or hardware vendors, are surprisingly buggy. Others work, but they aren't fully compatible with common test automation tools.

- Failure to use source control tools creates characteristic bugs. For example, if a bug goes away, comes back, goes away, comes back, goes away, comes back, then ask how the programming staff makes sure it's linking the most recent version of each module when it builds a version for you to test.

Software Errors:
Why are there Errors?

- Some programs or tasks are inherently complex. Boris Beizer talks perceptively about the locality problem in software. Think about an underlying bug, and then about symptoms caused by the bug. When symptoms appear, there's no assurance that they'll be close in time, space, or severity to the underlying bug. They may appear much later, or when working with a different part of the program, and they may seem much more or much less serious than the bug.

- Some programmers (some platforms) work with poor tools. Weak compilers, style checkers, debuggers, profilers, etc. make it too easy to get bugs or too hard to find bugs.

- Similarly, some third party hardware, or its drivers, are non-standard and don't respond properly to standard system calls. Incompatibility with hardware is often cited as the largest single source of customer complaints into technical support groups.

- When one programmer tries to fix bugs, or otherwise modify another programmer's code, there's lots of room for miscommunication and error.

- And, sometimes people just make mistakes.

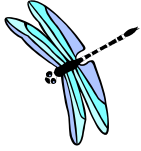# *Quality: Family Drug Store v. Gulf States Computer*

(563 So.2d 1324, Louisiana Court of Appeal, 1990). *The basic holding of this case is that a computer program that is honestly marketed can be extremely awkward to use without imposing liability on the seller.*

Two pharmacists bought a computer program known as the Medical Supply System from Gulf States. After they realized what they had bought, they asked for, and then sued for, a refund. Here were some of the problems of the system:

"1 all data had to be printed out, and could not be viewed on the monitor;

2 the information on the monitor would appear in code;

3 numerical codes were needed in order to open a new patient file

4 the system was unable to scroll."

The court found that the seller had not in any way misrepresented the system, and that it was not useless even though it was awkward to use. Further, the price of the software was about $2500 compared to $10,000 for other packages. *The plaintiffs had gotten what they'd paid for.*

Notes

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Decision Making, Information Flow, and Credibility

The Signal Detection & Recognition Problem

Response

Actual event	Bug (Response)	Feature (Response)
Bug	Hit	Miss
Feature	False Alarm	Correct Rejection

Refer to Testing Computer Software, pages 24, 116-118

Lessons From Signal Detection:
We Make Decisions Under Uncertainty

When you try to decide whether an item belongs to one category or the other (bug or feature), your decision will be influenced by your expectations and your motivation.

- Can you cut down on the number of false alarms without increasing the number of misses?

- Can you 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.

Lessons From Signal Detection: Decisions Are Subject To Bias

We make decisions under uncertainty.

Decisions are subject to bias, and much of this is unconscious.

The prime biasing variables are:

- *perceived probability:*

  If you think that an event is unlikely, you will be *substantially* less likely (beyond the actual probability) to report it.

- *perceived 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:*

  The degree to which you care / don't care can affect your willingness to adopt a decision rule that you might otherwise be more skeptical about.

Lessons From Signal Detection: Decisions Are Subject To Bias

Decisions are made by a series of people.

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

Trace these factors through the decisions and decision-makers (next slides). For example, what happens to your reputation if you

- Report every bug, no matter how minor, in order to make sure that 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?

Decisions Made During Bug Processing

Bug handling involves 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?

Your decisions will reflect on you. They will cumulatively have an effect on your credibility, because they reflect your judgment.

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

Refer to Testing Computer Software, pages 90-97, 115-118

Decisions Made
During Bug Processing-2

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

Programmer:

- Should I fix this bug or defer it?

Project Manager:

- Should I approve the deferral of this bug?

Tester:

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

Test Group Manager:

- Should I make an issue about this bug?
- 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?

Refer to Testing Computer Software, pages 90-97, 115-118

Decisions Made
During Bug Processing - 3

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 call for extra work in the answer book / advertising / manual / help?

***Director, Vice President, other senior staff*:**

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

Decisions Made During Bug Processing - 4

Who else is in your decision loop?

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

Issues That Will Bias People
Who <u>Evaluate</u> Bug Reports

These <u>reduce</u> the probability that the bug will be taken seriously and fixed.

- **Language critical of the programmer.**

- **Severity inflation.**

- **Pestering & refusing to ever take "No" for an answer.**

- **Tight schedule.**

- **Incomprehensibility, excessive detail, or apparent narrowness of the report.**

- **Weak reputation of the reporter.**

Issues That Will Bias People Who <u>Evaluate</u> Bug Reports

These <u>increase</u> the probability that the bug will be taken seriously and fixed.

- **Reliability requirements in this market.**
- **Ties to real-world applications.**
- **Report from customer/beta rather than from development.**
- **Strong reputation of the reporter.**
- **Weak reputation of the programmer.**
- **Poor quality/performance comparing to competitive product(s).**
- **News of litigation in the press.**

Clarify Expectations

One of the important tasks of a test manager is to clarify everyone's understanding of the use of the bug tracking database and to facilitate agreements that this approach is acceptable to the stakeholders.

- 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?

Make the rules explicit.

Biasing People Who <u>Report</u> Bugs

These will <u>reduce</u> the probability that bugs will be reported, by discouraging reporters, by convincing them that their work is pointless or will be filtered out, or by creating incentives for other people to pressure people not to report bugs.

- Never use bug statistics for employee bonus or discipline.
- Never use bug statistics to embarrass people.
- Never filter reports that you disagree with.
- Never change an in-house bug reporter's language, or at least not without free permission. Add your comments as additional notes, not as replacement text.
- Monitor language in the reports that is critical of the programmer or the tester.
- Beware of accepting lowball estimates of bug probabilities.

Biasing People Who <u>Report</u> Bugs

These help <u>increase</u> the probability that people will report bugs.

- Give results feedback to non-testers who report bugs.

- Encourage testers to report all anomalies.

- Adopt a formal system for challenging bug deferrals.

- Weigh schedule urgency consequences against an appraisal of quality costs. (Early in the schedule, people will report more bugs; later people will be more hesitant to report minor problems).

- Late in the schedule, set up a separate database for design issues (which will be evaluated for the start of the next release).

Notes

Bug Reporting Practice Example

You will be given a copy of a computer screen.

- Please write a description (all words, no pictures) of that screen

- When asked, please pass your description to your partner

- You will receive a description of a different screen from your partner.

- Please draw the screen that your partner is describing.

Sample Exam Question

**Give three different definitions of "software error."
Which do you prefer? Why?**

Sample Exam Question

Suppose that you find a reproducible failure that doesn't look very serious.

- Describe three tactics for testing whether the defect is more serious than it first appeared.

- As a particular example, suppose that the display got a little corrupted (stray dots on the screen, an unexpected font change, that kind of stuff) in the TI program when you entered data into a matrix. Describe some follow-up tests that you would run.

Notes

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____