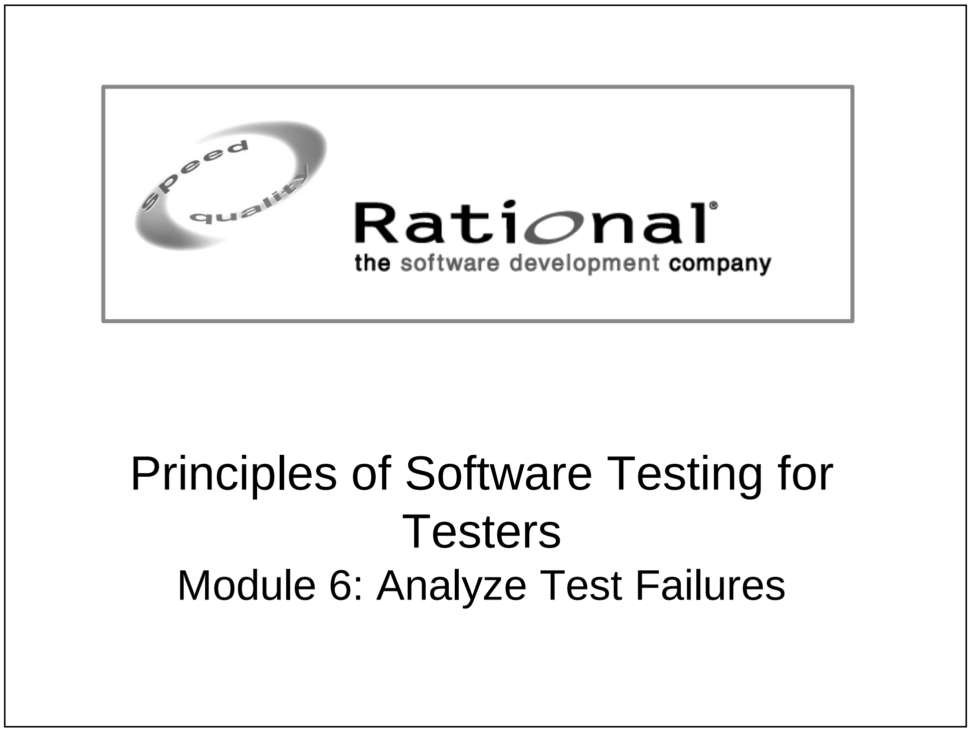
Module 6
Analyze Test Failures

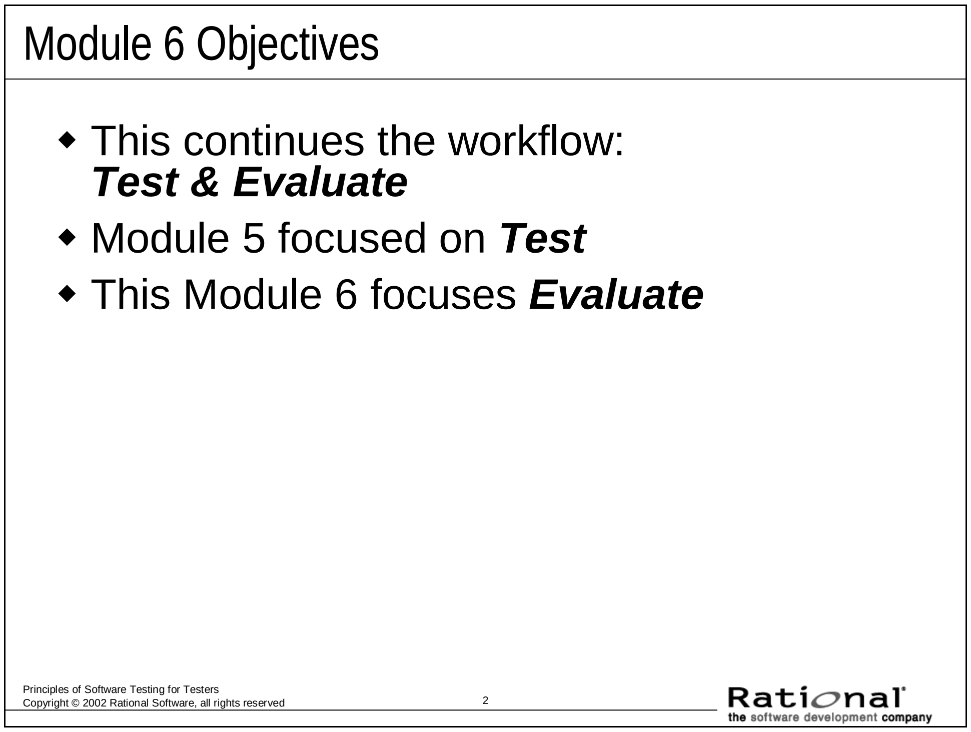


Rational®
the software development company

Principles of Software Testing for Testers
Module 6: Analyze Test Failures

Topics

Objectives



Module 6 Objectives

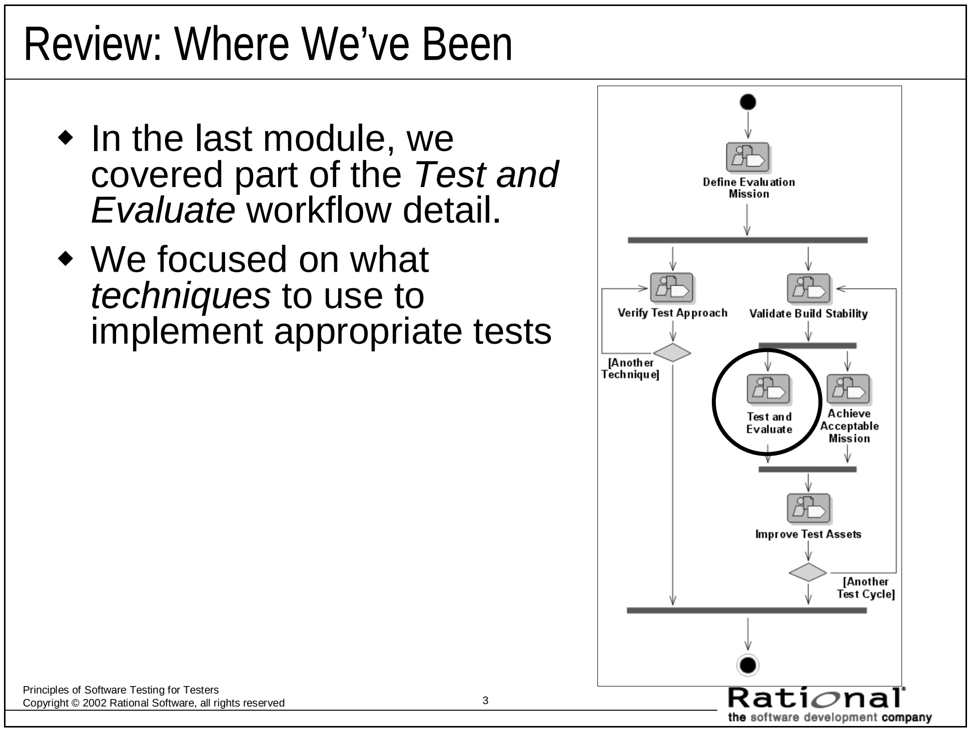
- This continues the workflow: ***Test & Evaluate***
- Module 5 focused on ***Test***
- This Module 6 focuses ***Evaluate***

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 2

Rational
the software development company

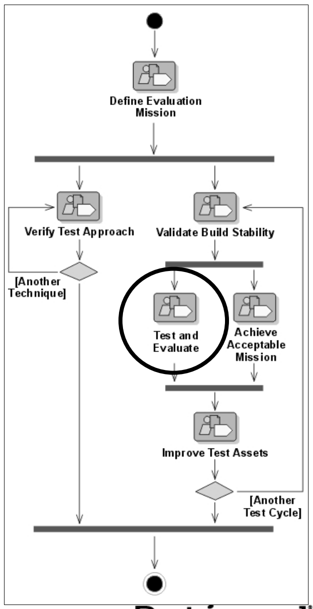
This module is devoted to a single RUP activity: *Analyze Test Failure*. The primary output of this activity is a Change Request. Making your change requests effective is one of the most important skills that you as a tester can develop.

Review: Where We've Been



Review: Where We've Been

- In the last module, we covered part of the *Test and Evaluate* workflow detail.
- We focused on what *techniques* to use to implement appropriate tests

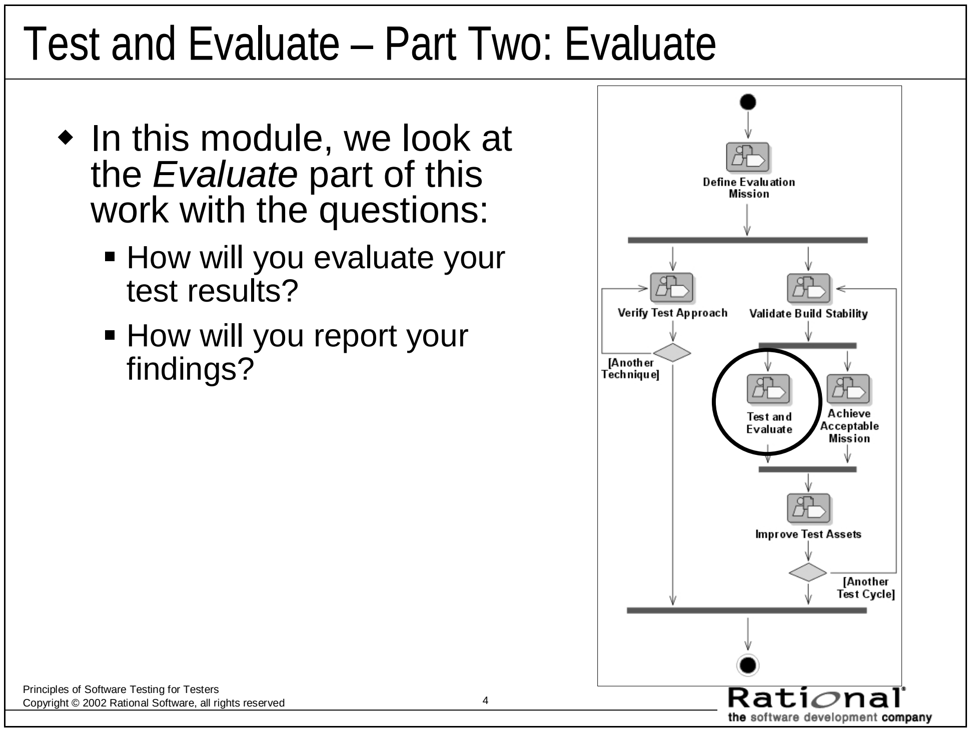


Principles of Software Testing for Testers

3

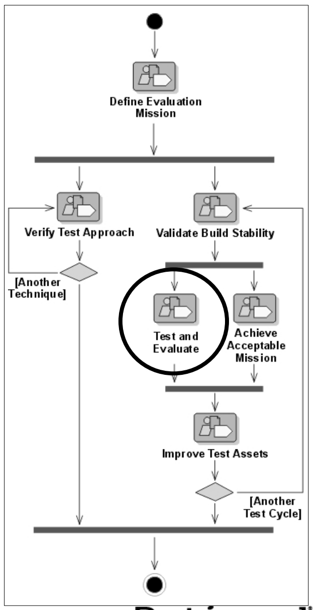
Rational the software development company

Test and Evaluate Workflow: Evaluate



Test and Evaluate – Part Two: Evaluate

- In this module, we look at the *Evaluate* part of this work with the questions:
  - How will you evaluate your test results?
  - How will you report your findings?



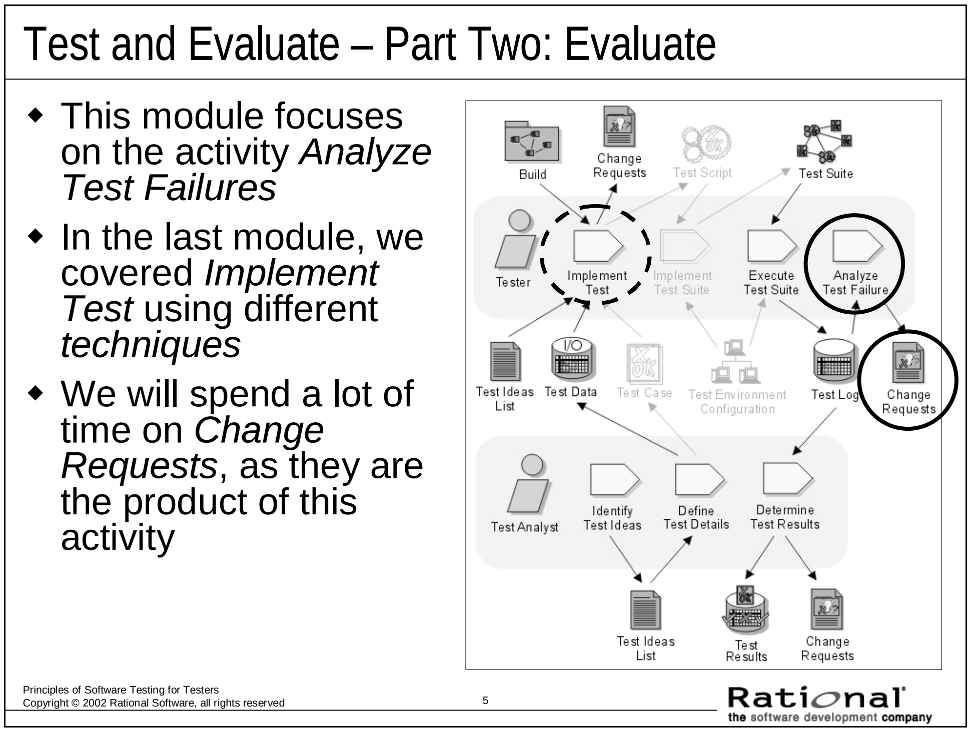
The purpose of this workflow detail is to achieve appropriate breadth and depth of the test effort to enable a sufficient evaluation of the Target Test Items — where sufficient evaluation is governed by the Test Motivators and Evaluation Mission.

For each test cycle, this work is focused mainly on:

- Providing ongoing evaluation and assessment of the Target Test Items
- Recording the appropriate information necessary to diagnose and resolve any identified Issues
- Achieving suitable breadth and depth in the test and evaluation work
- Providing feedback on the most likely areas of potential quality risk

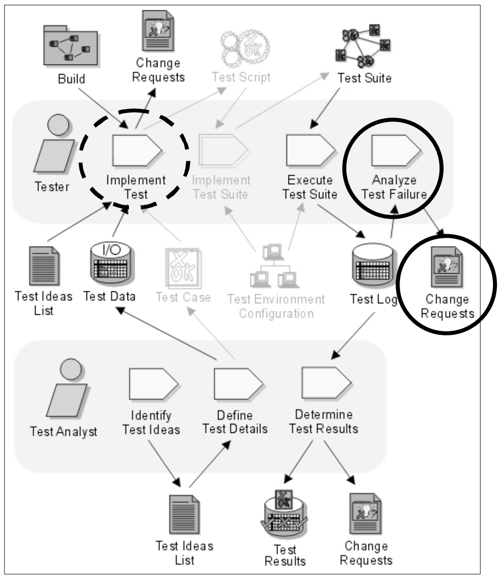
Typically enacted once per test cycle, this work involves performing the core tactical work of the test and evaluation effort: — namely the implementation, execution and evaluation of specific tests and the corresponding reporting of specific incidents that are encountered.

Test and Evaluate – Part Two: Evaluate



Test and Evaluate – Part Two: Evaluate

- This module focuses on the activity *Analyze Test Failures*
- In the last module, we covered *Implement Test* using different *techniques*
- We will spend a lot of time on *Change Requests*, as they are the product of this activity



Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved
5

Rational
the software development company

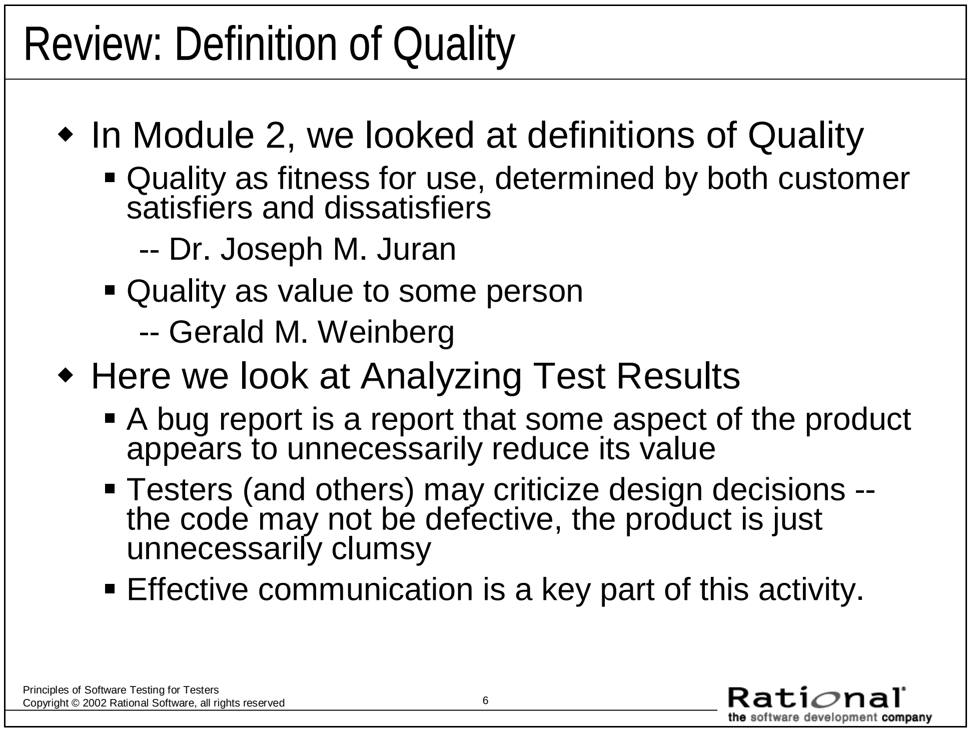
Here are the roles, activities and artifacts RUP focuses on in this work.

In the previous module, we discussed a selection of techniques that can be used to apply to a list of test ideas.

In this module, we'll talk more about evaluating the output of the tests that have been run.

Note that diagram shows some lightly shaded elements: these are additional testing elements that RUP provides guidance for which are not covered directly in this course. You can find out more about these elements by consulting RUP directly.

Review: Definition of Quality



Review: Definition of Quality

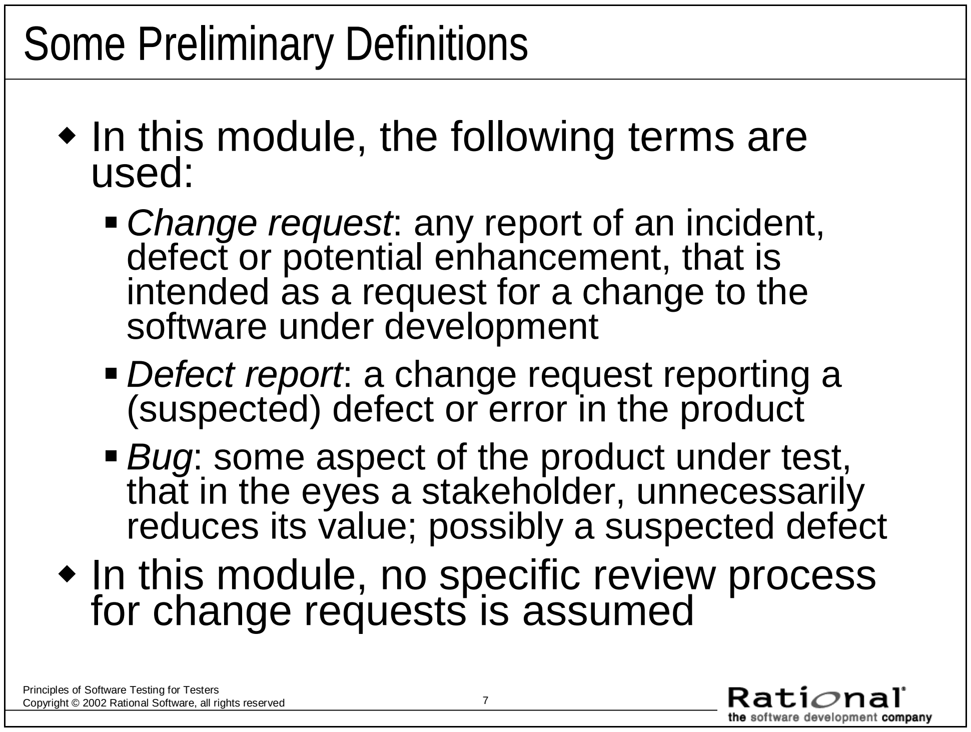
- In Module 2, we looked at definitions of Quality
  - Quality as fitness for use, determined by both customer satisfiers and dissatisfiers
    -- Dr. Joseph M. Juran
  - Quality as value to some person
    -- Gerald M. Weinberg
- Here we look at Analyzing Test Results
  - A bug report is a report that some aspect of the product appears to unnecessarily reduce its value
  - Testers (and others) may criticize design decisions -- the code may not be defective, the product is just unnecessarily clumsy
  - Effective communication is a key part of this activity.

Principles of Software Testing for Testers

6
Rational the software development company

In discussion of the analysis, we have to refer back to the concept of quality that we are applying.

Some Preliminary Definitions



Some Preliminary Definitions

- In this module, the following terms are used:
  - *Change request*: any report of an incident, defect or potential enhancement, that is intended as a request for a change to the software under development
  - *Defect report*: a change request reporting a (suspected) defect or error in the product
  - *Bug*: some aspect of the product under test, that in the eyes a stakeholder, unnecessarily reduces its value; possibly a suspected defect
- In this module, no specific review process for change requests is assumed

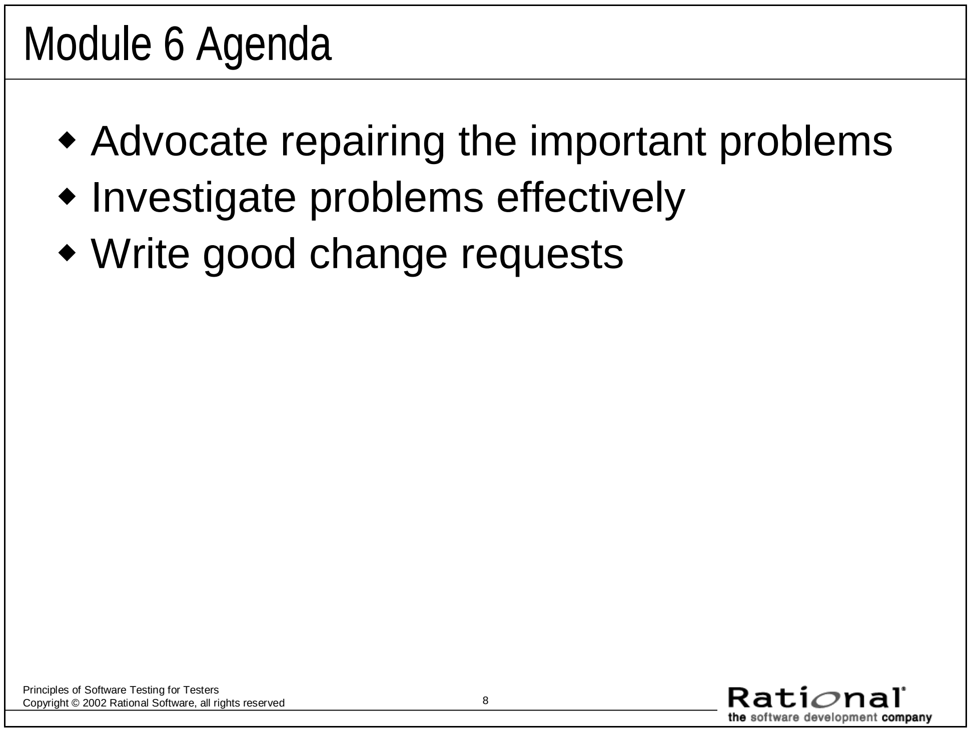
Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 7

Rational
the software development company

In some companies, there is high sensitivity to the choice of words among *change*, *defect*, *incident*, and *bug*. Often there are very good reasons for this sensitivity, such as liability, contractual obligation or professional culture. In this course, we use the terms *defect* and *bug* interchangeably, and correspondingly *defect report* and *bug report*.

Processes vary across companies and projects. Accordingly, it is sometimes project managers who review and assign defects, sometimes a triage committee, sometimes a Change Control Manager, sometimes a Change Control Board, sometimes a developer directly. For purposes of this module, these differences do not matter. In every case, we assume that you, as a tester, write your defect report for an intended audience similar to one of these.

Module 6 Agenda



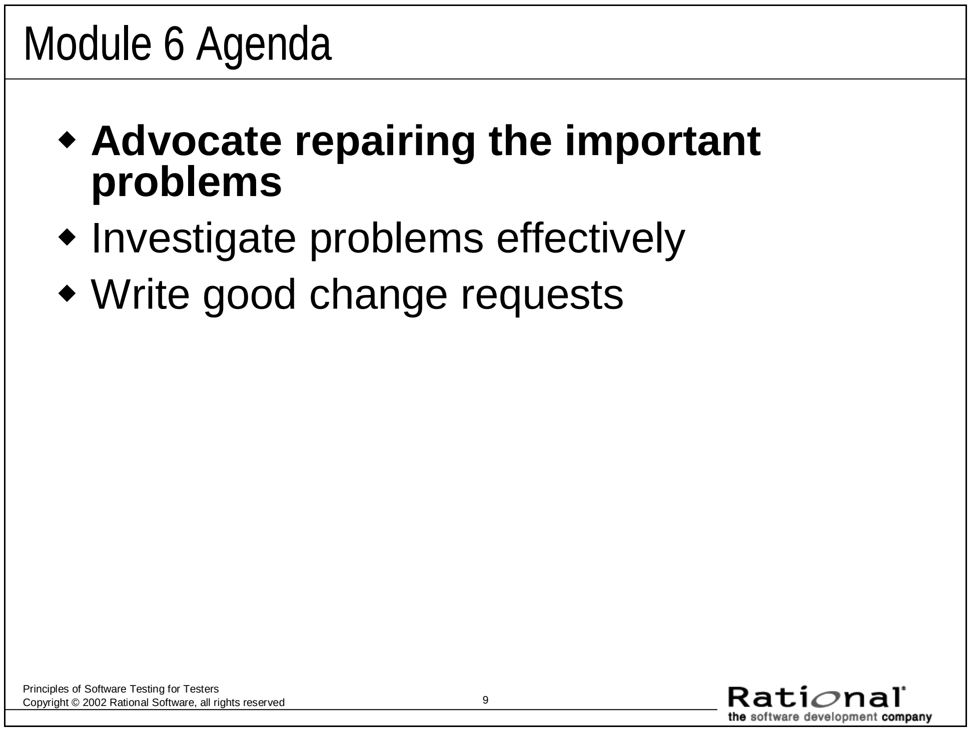
Module 6 Agenda

- Advocate repairing the important problems
- Investigate problems effectively
- Write good change requests

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 8

Rational® the software development company

Advocate Repairing the Important Problems



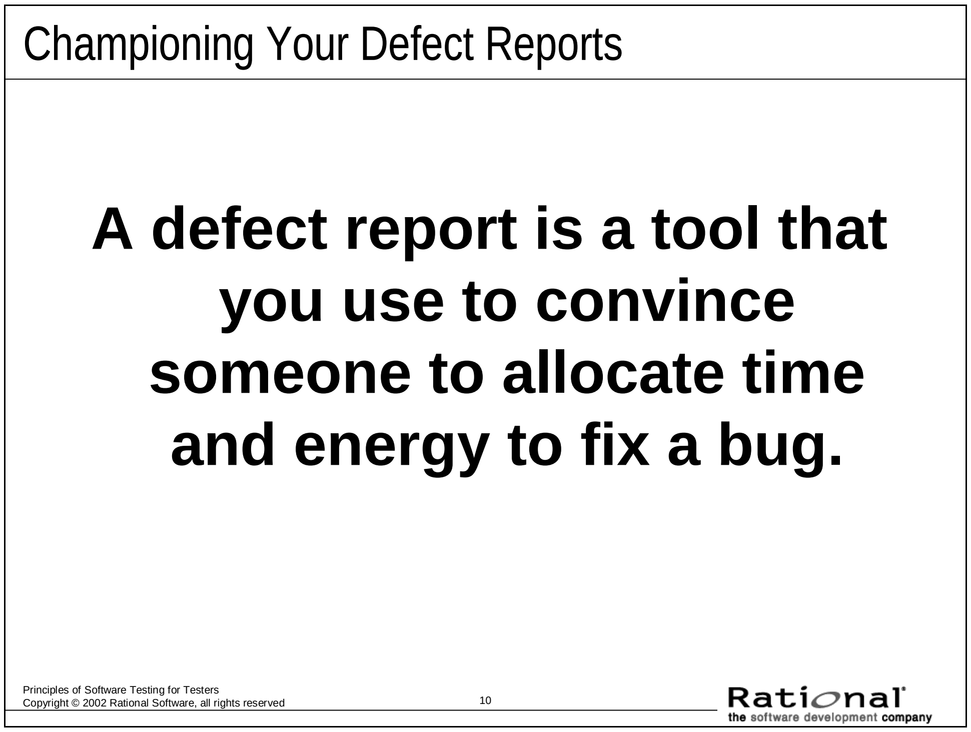
Module 6 Agenda

- **Advocate repairing the important problems**
- Investigate problems effectively
- Write good change requests

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 9

Rational the software development company

Championing Your Defect Reports



Championing Your Defect Reports

A defect report is a tool that you use to convince someone to allocate time and energy to fix a bug.

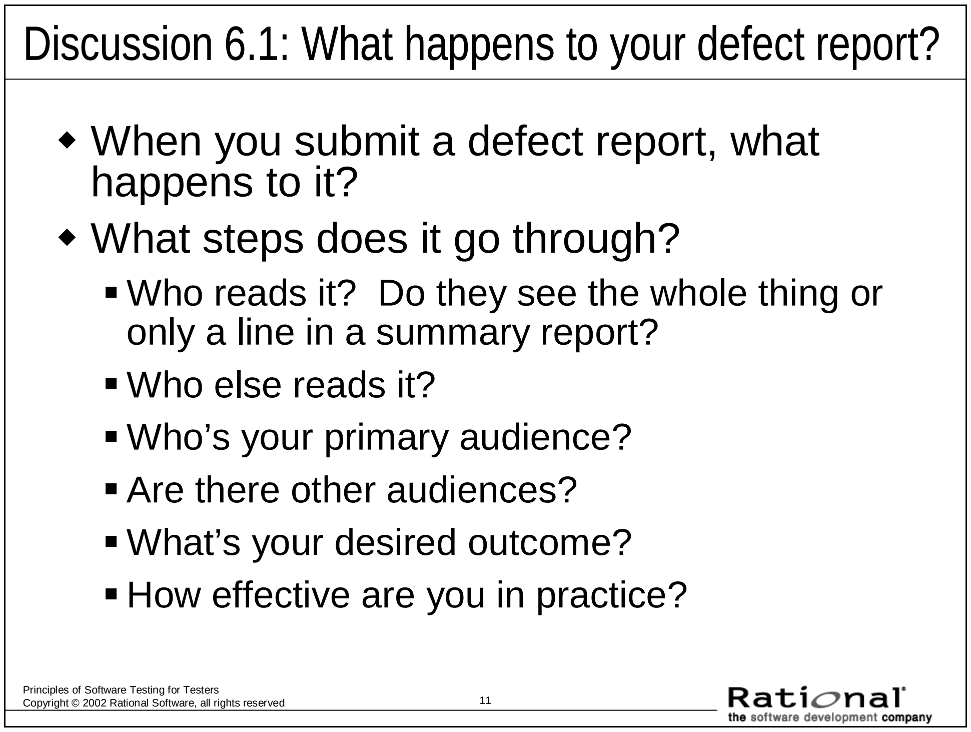
This leads to two important points.

1. **Visibility**. Defect reports are most testers' primary work product. Your defect reports are the only thing that many managers know about you.

> *You get a reputation based on what you write*. This is what people outside of the test group will most notice and most remember of your work.

2. **Effectiveness**. The best tester isn't the one who finds the most bugs or who embarrasses the most developers. *The best tester is the one who gets the most bugs fixed*.

Discussion 6.1: What happens to your defect report?



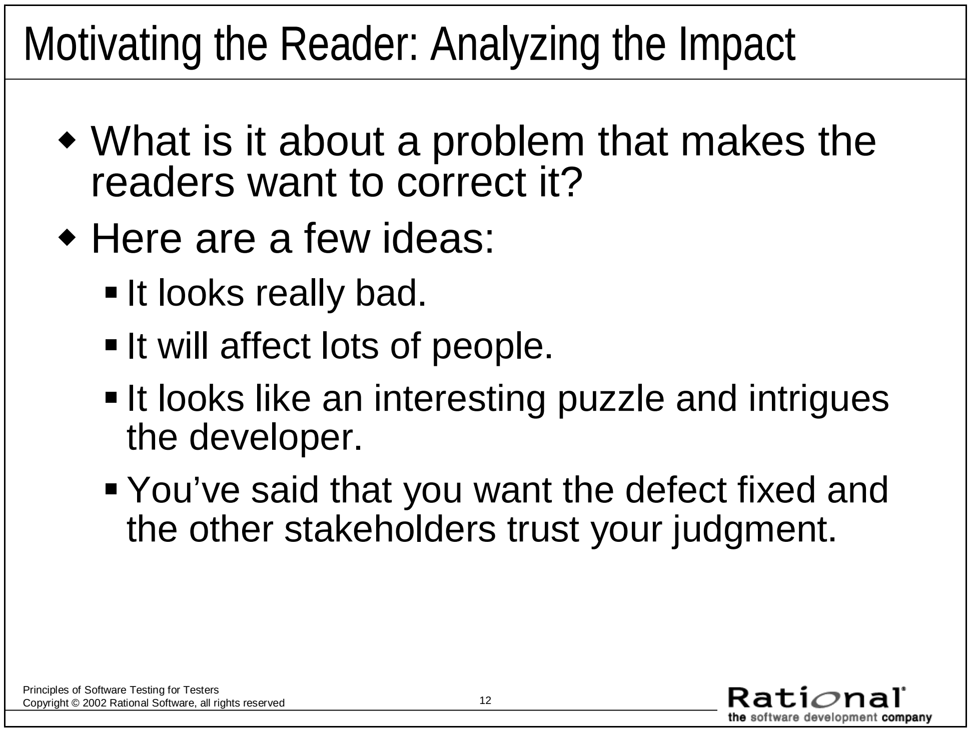
Discussion 6.1: What happens to your defect report?

- When you submit a defect report, what happens to it?
- What steps does it go through?
  - Who reads it? Do they see the whole thing or only a line in a summary report?
  - Who else reads it?
  - Who's your primary audience?
  - Are there other audiences?
  - What's your desired outcome?
  - How effective are you in practice?

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 11

Optional Exercise

Motivating the Reader: Analyzing the Impact



Motivating the Reader: Analyzing the Impact

- What is it about a problem that makes the readers want to correct it?
- Here are a few ideas:
  - It looks really bad.
  - It will affect lots of people.
  - It looks like an interesting puzzle and intrigues the developer.
  - You've said that you want the defect fixed and the other stakeholders trust your judgment.

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 12

Rational
the software development company

For additional discussion, see Chapter 4 of *Lessons Learned*.

What We Can Learn From a Sales Model

You write defect reports for developers and managers whose time is in short supply. If you want to convince the people who determine whether a defect will be worked on to allocate time to investigate and fix a bug, you may have to sell them on it.

In this sense, selling the project team on a defect revolves around two fundamental objectives:

Motivate the buyer - Make him WANT to fix the bug.

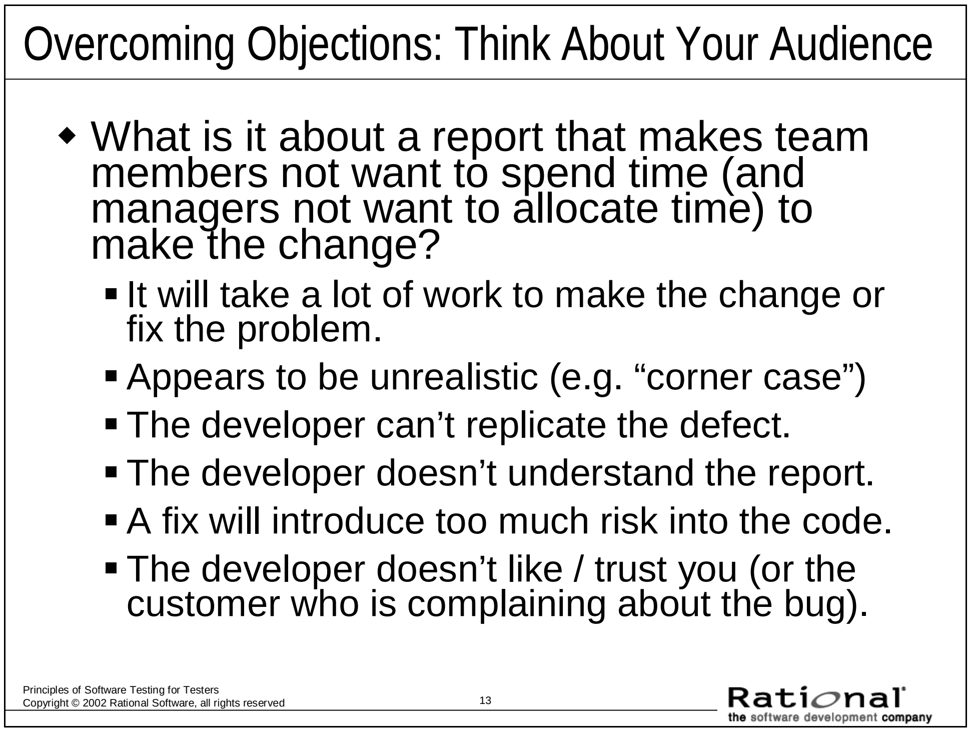
Overcome objections - Address his reasons (both valid concerns and excuses) for not fixing the bug.

More Ideas on Motivating the Developer

Here are a few more examples of ours. You might think of additional examples, that are particularly relevant in your company.

- 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.
- A trade magazine recently criticized another company for a very similar case.

Overcoming Objections: Think About Your Audience



Overcoming Objections: Think About Your Audience

- What is it about a report that makes team members not want to spend time (and managers not want to allocate time) to make the change?
  - It will take a lot of work to make the change or fix the problem.
  - Appears to be unrealistic (e.g. "corner case")
  - The developer can't replicate the defect.
  - The developer doesn't understand the report.
  - A fix will introduce too much risk into the code.
  - The developer doesn't like / trust you (or the customer who is complaining about the bug).

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 13

Rational the software development company

Here are some more examples:

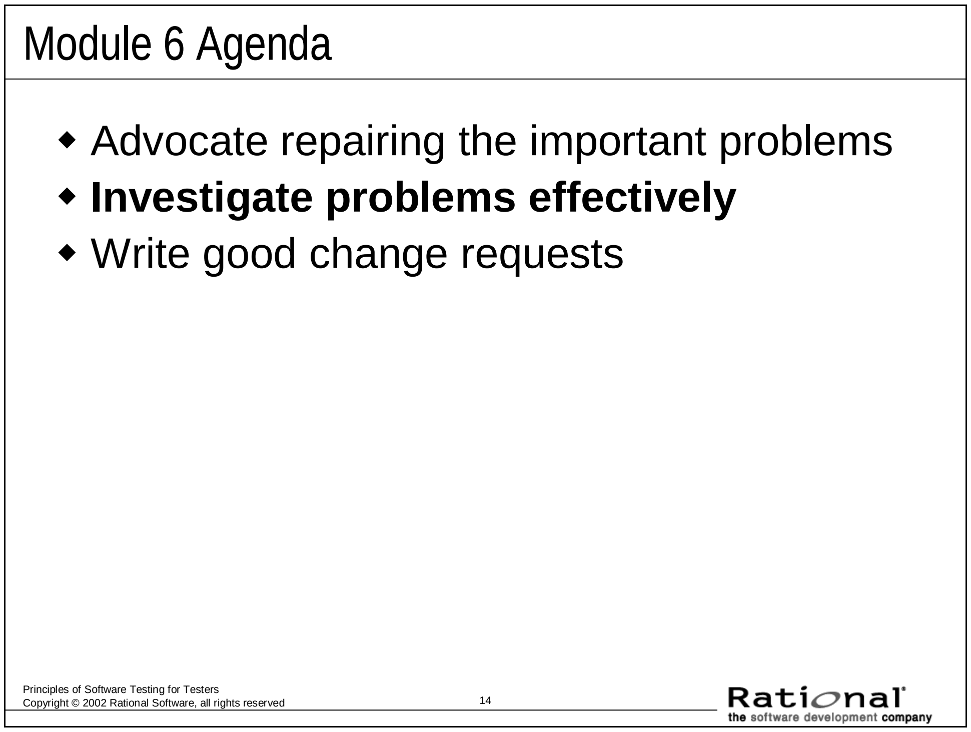
- Strange and complex set of steps required to induce the failure.
- Not enough information to know what steps are required, and it will take a lot of work to figure them out.
- No perceived customer impact
- Unimportant (no one will care if this is wrong: minor error or unused feature.)
- That's not a bug, it's a feature.
- Management doesn't care about bugs like this.

These are sometimes valid responses and are often excuses. Bugs that "no one would care about" are often more significant than the developers think.

Testers can often avoid these objections by doing better research and writing defect reports that are designed to be less vulnerable to objection.

We'll look at methods to avoid or overcome objections in the set of slides after the slides on follow-up testing.

Investigate Problems Effectively



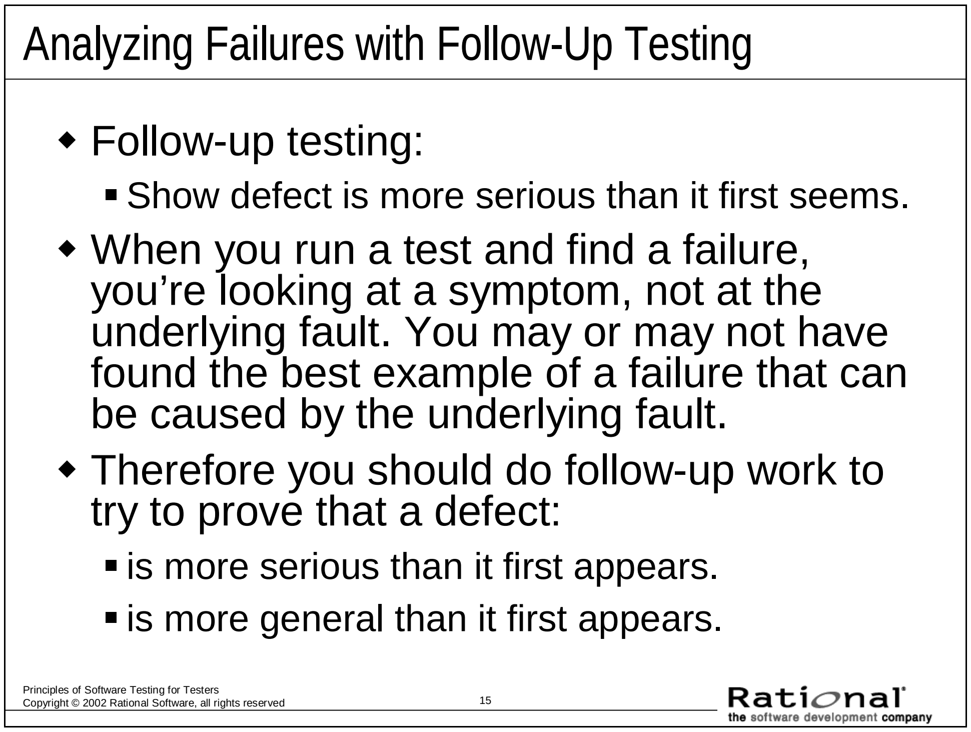
Module 6 Agenda

- Advocate repairing the important problems
- **Investigate problems effectively**
- Write good change requests

14

Rational
the software development company

Analyzing Failures with Follow-Up Testing



Analyzing Failures with Follow-Up Testing

- Follow-up testing:
  - Show defect is more serious than it first seems.
- 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 follow-up work to try to prove that a defect:
  - is more serious than it first appears.
  - is more general than it first appears.

You see a problem. It looks relatively minor. Should you stop there or should you investigate further to see if you can find a more serious version of the same problem?

After all, the failure is just a symptom of an underlying error in the program. If you apply slightly different conditions to the program, you might see a much worse result.

Question: Is it deceptive or misleading to present the bug in its most powerful form? ABSOLUTELY NOT. To balance the risks properly, the project team needs to understand how serious the problem is. If they see a weaker form of the bug, they will assess it incorrectly.

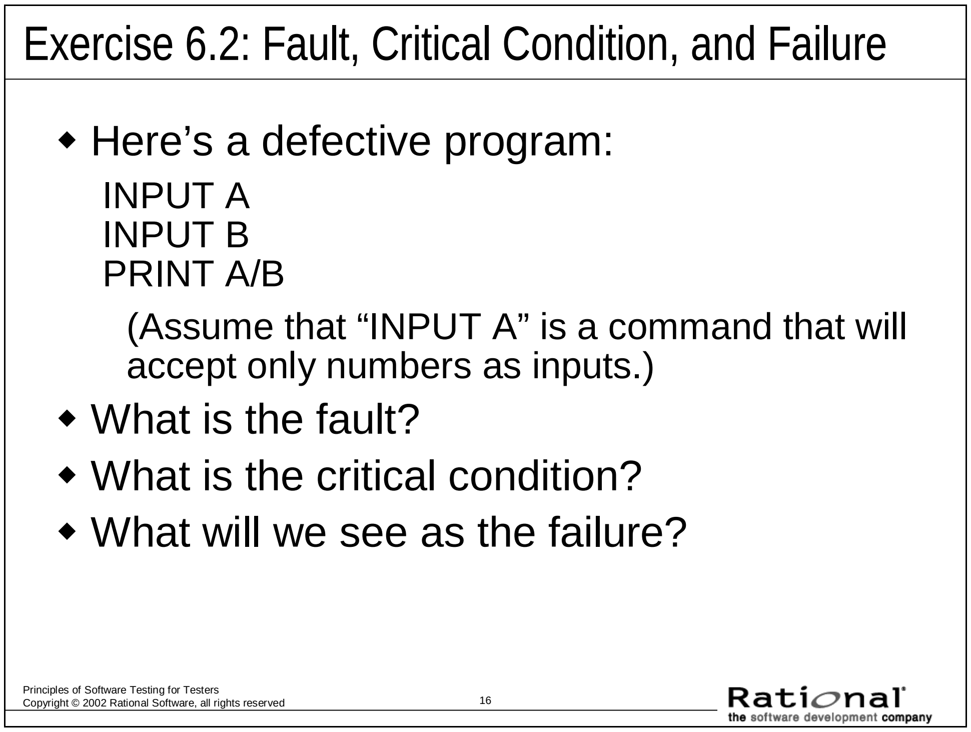
On the other hand, if you find a problem that looks minor under most conditions, but looks more serious in a special case, the report should make clear the differences in behavior under the different conditions.

- Start with the simplest description of the most direct set of steps that will take the developer to the most serious failure.
- But then, after you have clearly and simply laid out a good set of replication steps, add a set of notes that describe variations of the test that you ran and the results you obtained.

This is useful to most developers, for troubleshooting purposes. It can save them a lot of time.

Additionally, these notes will enhance or protect your credibility. The goal is not to deceive the developer into thinking the problem is more serious than it is. The goal is to catch the attention of the developer and make sure that he (and / or the project team) takes the problem seriously enough to prioritize it accurately.

Exercise 6.2: Fault, Critical Condition, and Failure



Exercise 6.2: Fault, Critical Condition, and Failure

- Here's a defective program:

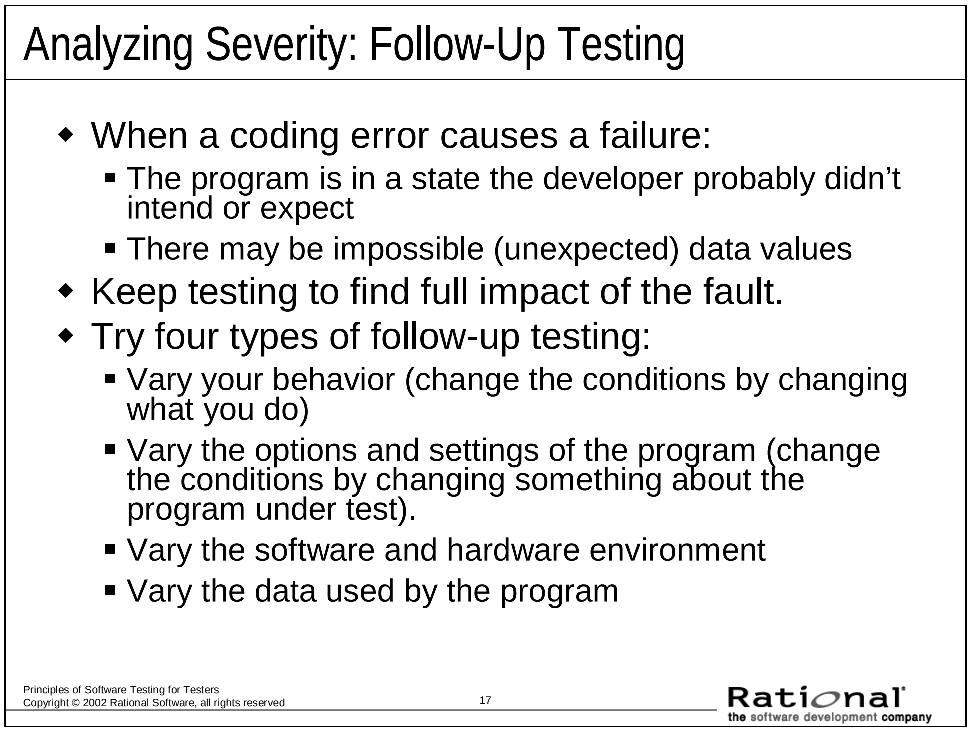
```
INPUT A
INPUT B
PRINT A/B
```

  (Assume that "INPUT A" is a command that will accept only numbers as inputs.)
- What is the fault?
- What is the critical condition?
- What will we see as the failure?

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 16

Optional Exercise

Analyzing Severity: Follow-Up Testing



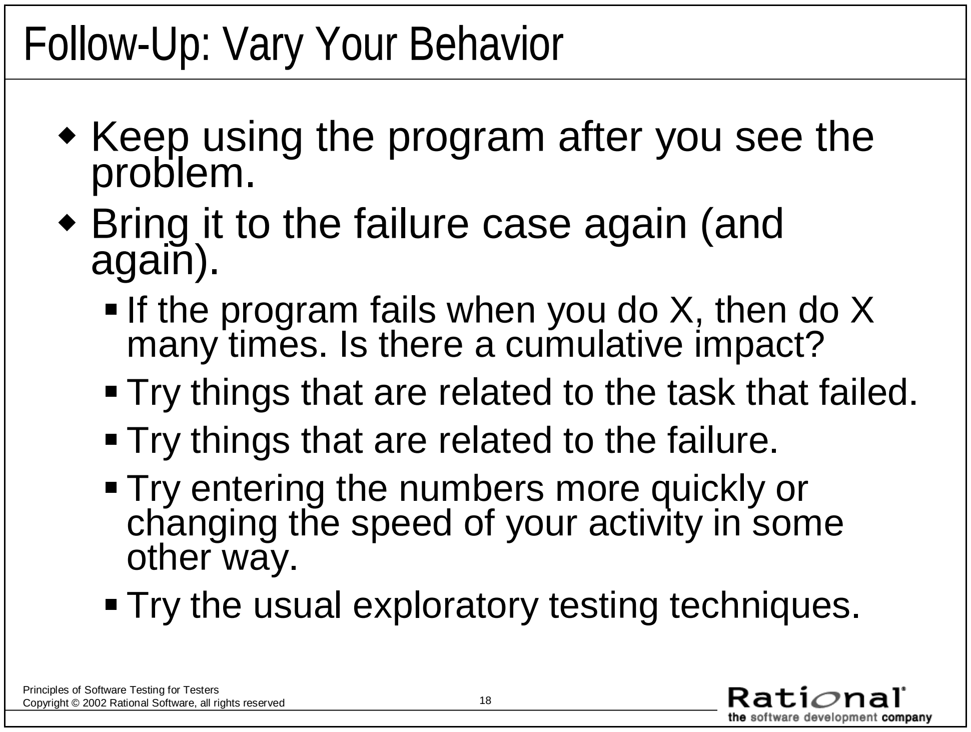
Analyzing Severity: Follow-Up Testing

- When a coding error causes a failure:
  - The program is in a state the developer probably didn't intend or expect
  - There may be impossible (unexpected) data values
- Keep testing to find full impact of the fault.
- Try four types of follow-up testing:
  - Vary your behavior (change the conditions by changing what you 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
  - Vary the data used by the program

The essence of great defect analysis is helping people understand the impact and where to fix. Follow-Up Testing is the basis for much of that analysis. It is a form of Exploratory Testing in which you investigate other effects of the fault. For example, if fast typing seems to trigger a problem, try a tool to type even faster or try cut and paste. If the defect seems related to a low-end configuration, use the lowest-spec configuration for testing.

Keeping a notebook for yourself is very helpful. Running tests with all these variations requires that you you can trace a result to the variation that caused it. The simplest method is often a lab notebook where you record the initial conditions with the result, then log each variation and the result of running with that particular variation.

Follow-Up: Vary Your Behavior



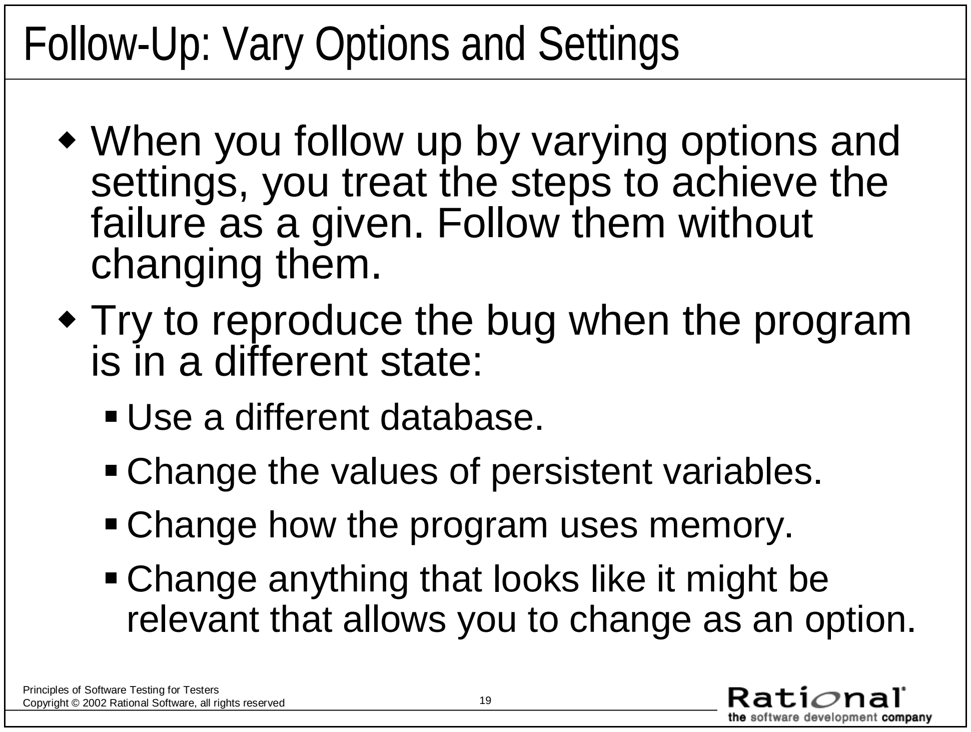
Follow-Up: Vary Your Behavior

- 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.
  - Try things that are related to the failure.
  - Try entering the numbers more quickly or changing the speed of your activity in some other way.
  - Try the usual exploratory testing techniques.

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 18

- 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.
- *Trying the usual exploratory testing techniques*. 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

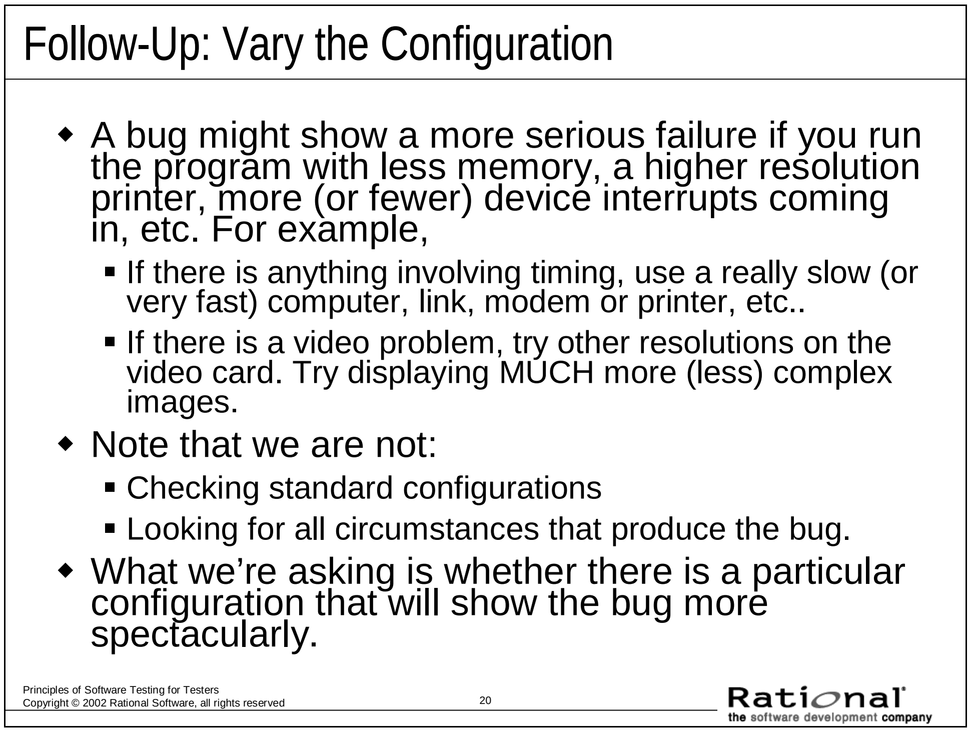


Follow-Up: Vary Options and Settings

- When you follow up by varying options and settings, you treat the steps to achieve the failure as a given. Follow them without changing them.
- 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



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. For example,
  - 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
  - Looking for all circumstances that produce the bug.
- What we're asking is whether there is a particular configuration that will show the bug more spectacularly.

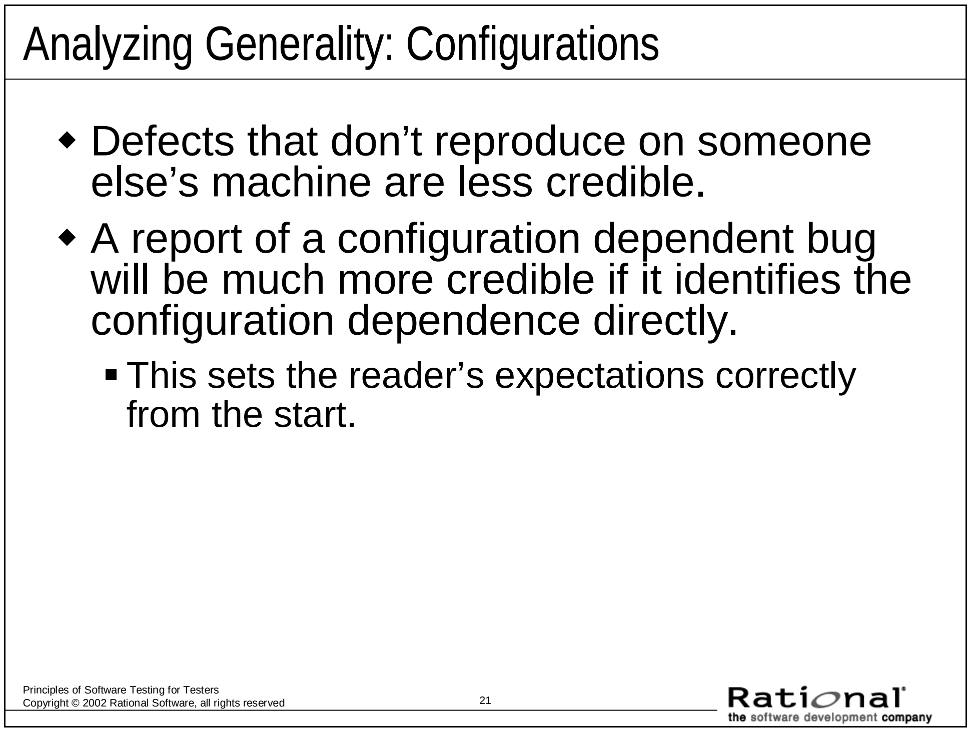
Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 20

Rational
the software development company

A more spectacular failure will generate more interest in getting the bug fixed. Returning to the example (unexpected scrolling when you add two numbers), try things like:

1. Different video resolutions
2. Different mouse settings if you have a wheel mouse that does semi-automated scrolling
3. A television signal output (e.g. NTSC, PAL, SECAM) instead of a traditional (XGA or SVGA, etc.) monitor output.

Analyzing Generality: Configurations



Analyzing Generality: Configurations

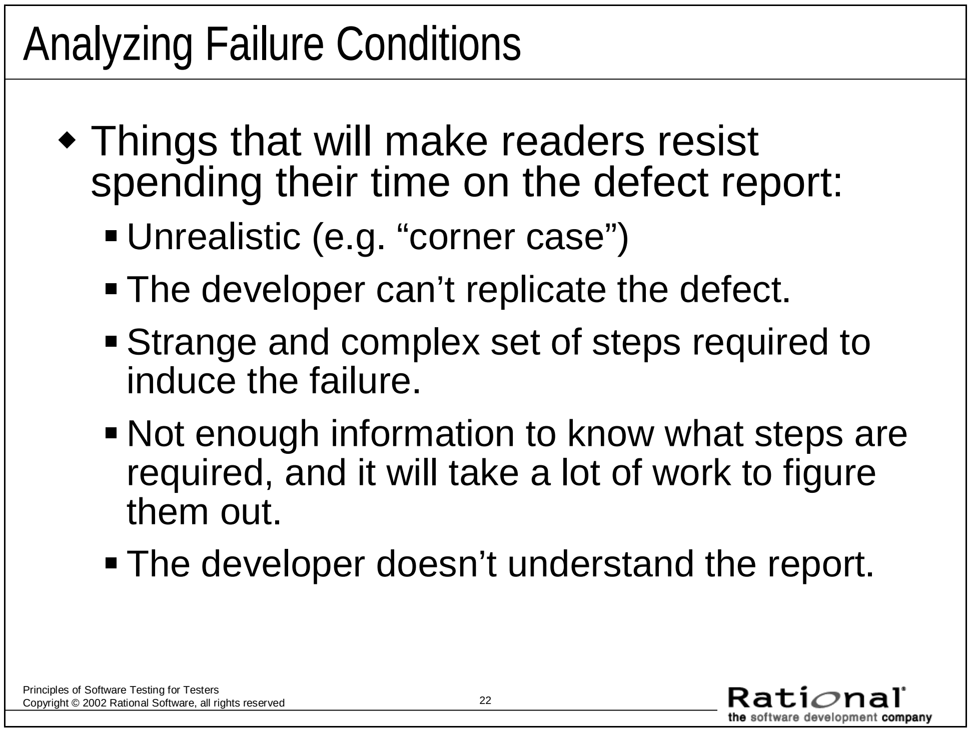
- Defects that don't reproduce on someone else's machine are less credible.
- A report of a configuration dependent bug will be much more credible if it identifies the configuration dependence directly.
  - This sets the reader's expectations correctly from the start.

In the ideal case, you should test on 2 machines. This is standard in many companies.

1. 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.
2. When you find a defect, use Machine 1 as your defect 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.
3. Some people do their main testing on the turtle and use the power machine for replication.
4. 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 defect report while you wrote it. (A valuable thing to do.)
5. 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.

Analyzing Failure Conditions



Analyzing Failure Conditions

- Things that will make readers resist spending their time on the defect report:
  - Unrealistic (e.g. “corner case”)
  - The developer 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 developer doesn’t understand the report.

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 22

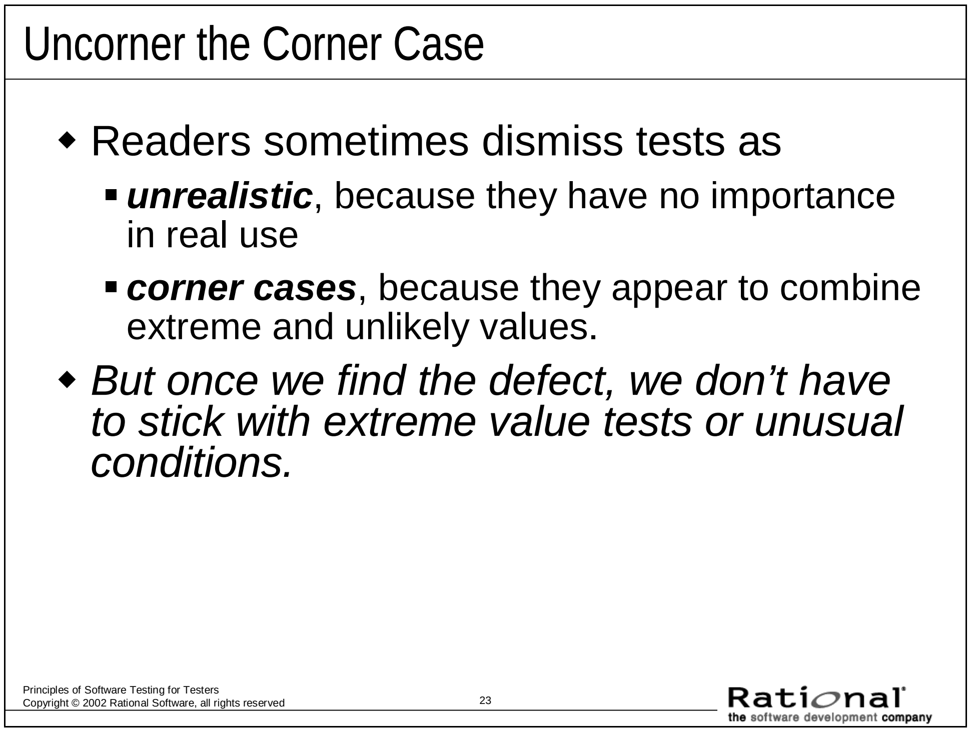
Rational the software development company

Up to now, we’ve been talking about researching failure conditions in order to make the report more powerful, in order to make the programmers and management more motivated to fix the problem.

We are now talking about researching the failure conditions to make the report less objectionable to the programmer than it otherwise might be.

Some of these “objections” sound like common excuses for not fixing the problem. Sometimes they are mere excuses; but they usually are based in some form of weakness of the report. You can make the report less vulnerable to these criticisms.

Uncorner the Corner Case



Uncorner the Corner Case

- Readers sometimes dismiss tests as
  - ***unrealistic***, because they have no importance in real use
  - ***corner cases***, because they appear to combine extreme and unlikely values.
- *But once we find the defect, we don't have to stick with extreme value tests or unusual conditions.*

Some reports are inevitably dismissed as **unrealistic**. You can do follow-up testing to either find the problem with mainstream values or isolate it to a narrow set of extreme values. If you're protesting a bug that has been left unfixed for several shipped versions, some people will believe no one cares about it. Perhaps, though, customer complaints about this bug have never filtered through to developers.

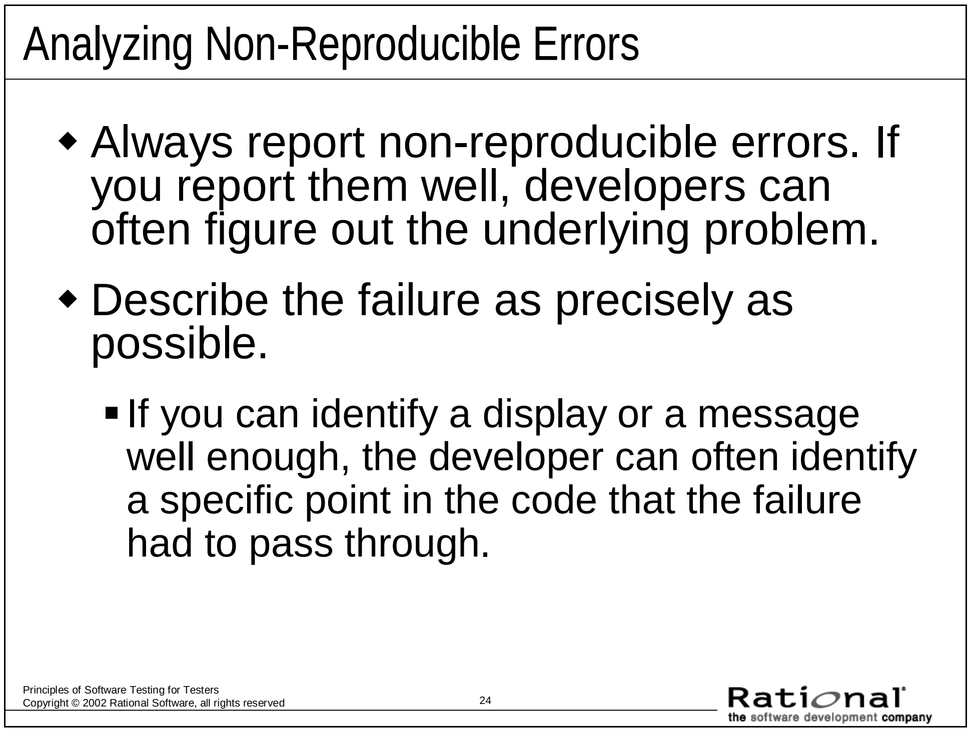
We test at **extreme values** because these are the most likely places to show a defect. If you find a problem, however, try mainstream values. These are the easy settings that should pose no problem to the program. If you can still replicate the problem, refer primarily to the mainstream settings when you write up the report . This will be a very credible defect report.

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

1. Is the bug tied to a single setting (a true corner case)?
2. Or is there a small range of cases? What is it?
3. 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.

If your report of a defect or design issue is dismissed as having "no customer impact," don't accept this at face value from a programmer or project manager. Ask yourself who might actually know the customer impact? Good people to try are Technical Marketing, Technical Support, Human Factors, Documentation, Network Administrators, Training, In-house Power Users, and maybe Sales.

Analyzing Non-Reproducible Errors



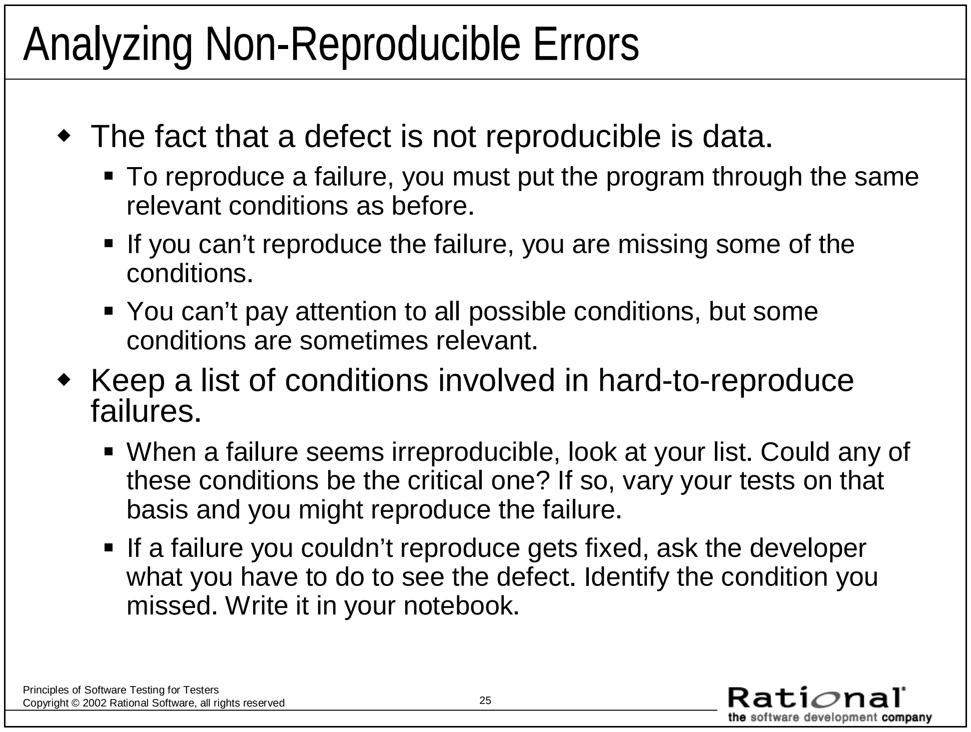
Analyzing Non-Reproducible Errors

- Always report non-reproducible errors. If you report them well, developers can often figure out the underlying problem.
- Describe the failure as precisely as possible.
  - If you can identify a display or a message well enough, the developer can often identify a specific point in the code that the failure had to pass through.

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved
24

- 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 developer***. Don't try to pass the bug on to the developer at this point. Instead say that you have a bug that you can't reproduce and you wonder if there are any ideas for follow-up tests that you can run.

Analyzing Non-Reproducible Errors



Analyzing Non-Reproducible Errors

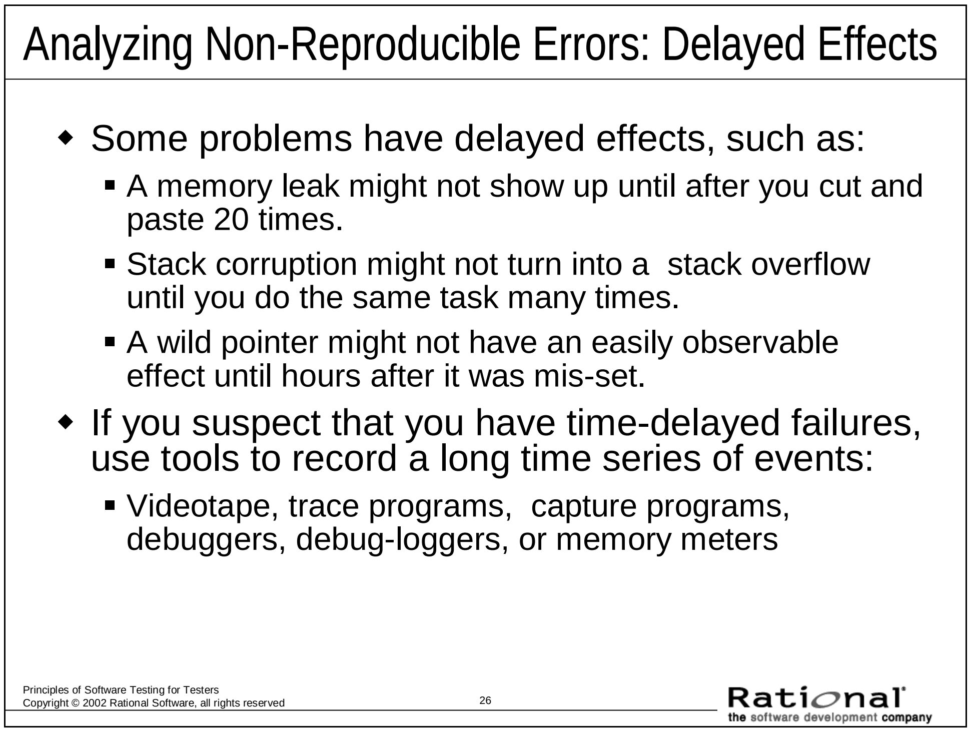
- The fact that a defect is not reproducible is data.
  - To reproduce a failure, you must put the program through the same relevant conditions as before.
  - If you can't reproduce the failure, you are missing some of the conditions.
  - You can't pay attention to all possible conditions, but some conditions are sometimes relevant.
- Keep a list of conditions involved in hard-to-reproduce failures.
  - When a failure seems irreproducible, look at your list. Could any of these conditions be the critical one? If so, vary your tests on that basis and you might reproduce the failure.
  - If a failure you couldn't reproduce gets fixed, ask the developer what you have to do to see the defect. Identify the condition you missed. Write it in your notebook.

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 25

Rational
the software development company

- Here are some examples of conditions that might be relevant in your environment.
- The bug depends on the value of a hidden input variable. (Bob Stahl teaches this well in his courses on testing.) 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 developer about what state variables or flags get set in the course of using a particular feature.

Analyzing Non-Reproducible Errors: Delayed Effects



Analyzing Non-Reproducible Errors: Delayed Effects

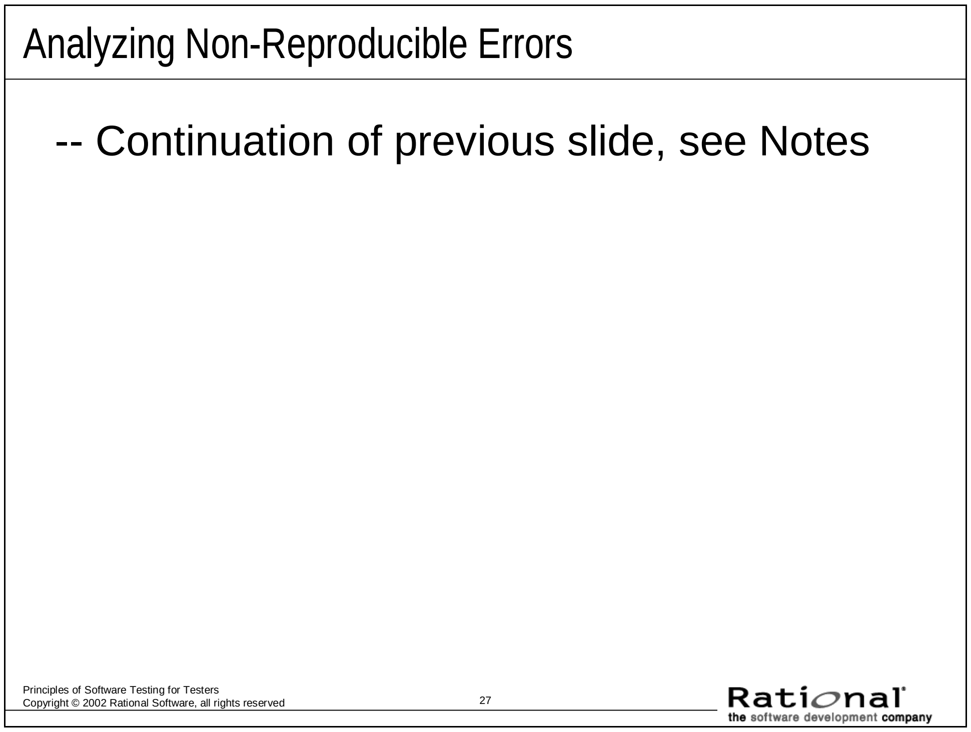
- Some problems have delayed effects, such as:
  - A memory leak might not show up until after you cut and paste 20 times.
  - Stack corruption might not turn into a stack overflow until you do the same task many times.
  - A wild pointer might not have an easily observable effect until hours after it was mis-set.
- If you suspect that you have time-delayed failures, use tools to record a long time series of events:
  - Videotape, trace programs, capture programs, debuggers, debug-loggers, or memory meters

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 26

Rational the software development company

- Some conditions are catalysts. They make failures more likely to be seen. Example: low memory for a leak; slow machine for a race condition. 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?
- The bug might appear only at a specific date or time of day. Look for week-end, month-end, quarter-end and year-end bugs.
- 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 analyze different types of coupling and discuss strengths and vulnerabilities that these can create.) Interrupts may 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 developer).
- The bug depends on you doing related tasks in a specific order.
- 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.

Analyzing Non-Reproducible Errors

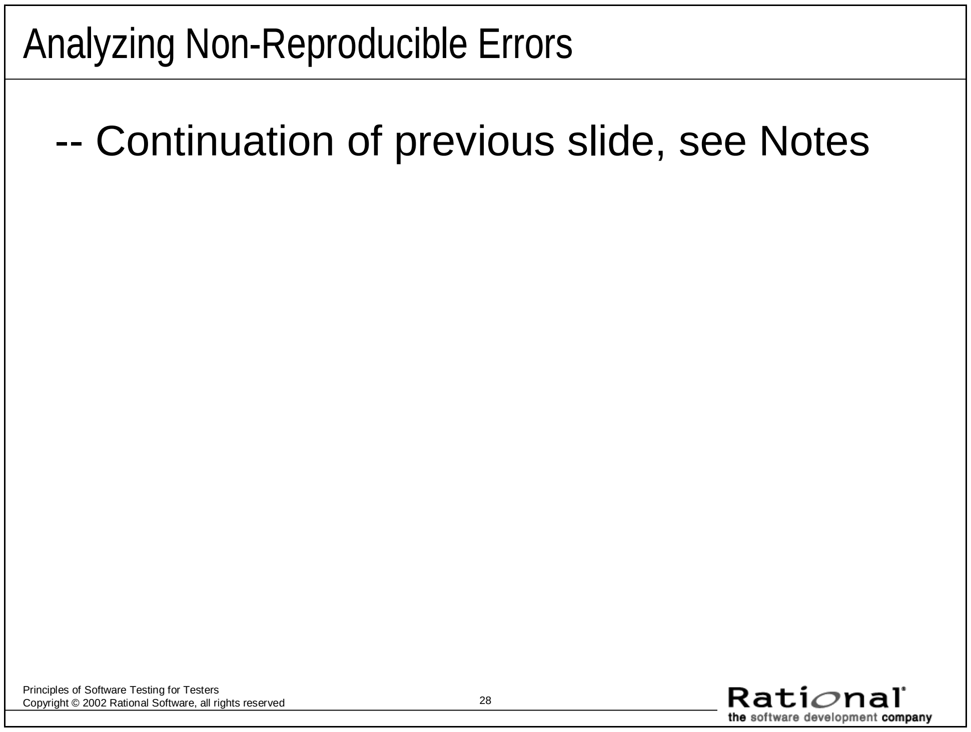


Analyzing Non-Reproducible Errors

-- Continuation of previous slide, see Notes

- 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 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?
- 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.
- 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.

Analyzing Non-Reproducible Errors

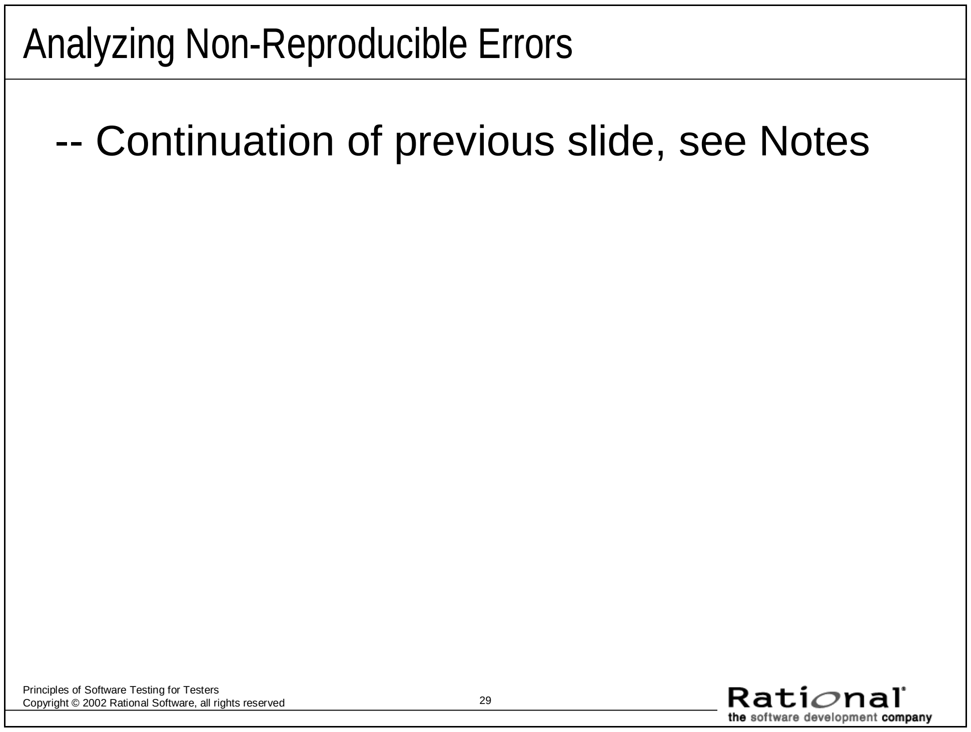


Analyzing Non-Reproducible Errors

-- Continuation of previous slide, see Notes

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

Analyzing Non-Reproducible Errors

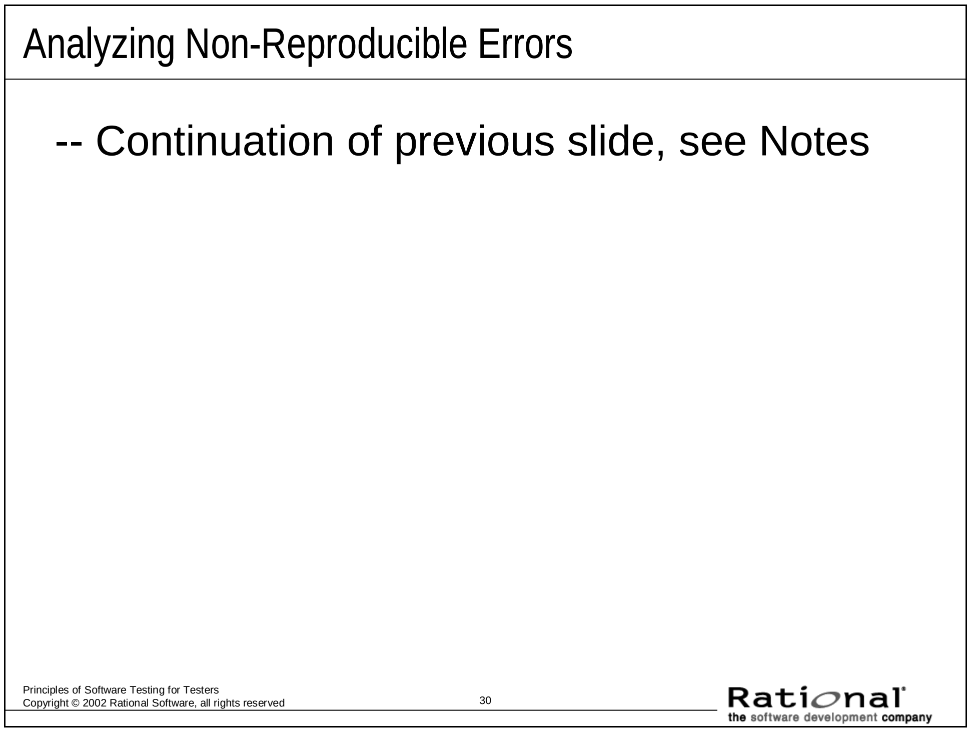


Analyzing Non-Reproducible Errors

-- Continuation of previous slide, see Notes

- 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.
- 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. (Just checking that you were reading these notes.)

Analyzing Non-Reproducible Errors



Analyzing Non-Reproducible Errors

-- Continuation of previous slide, see Notes

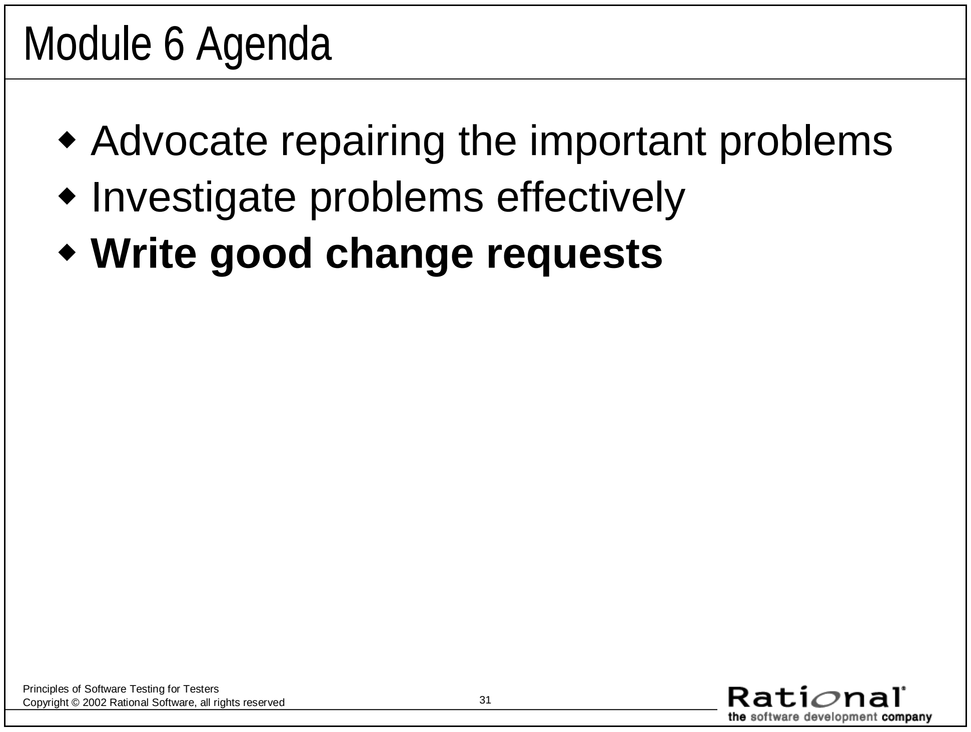
Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 30

Hung Quoc Nguyen describes several other ideas (focused on web testing) at http://www.logigear.com/testing/Issue2_3_Nguyen.pdf

Here's a sample of what he has to say:

> "When the value of a specific environment attribute does not stay constant each time a test procedure is executed, it causes the operating environment to become dynamic. The attribute can be anything from resource specific such as available RAM, disk space, etc., to timing specific such as network latency, the order of user-transactions being submitted, etc.
>
> When a test case depends on the exact replication of both, the set of steps and the operating environment but the operating environment cannot be replicated (due to its dynamic nature), the error becomes irreproducible or hard-to-reproduce.
>
> By the way, this is why memory related errors are often hard-to-reproduce. Take an example of a memory-overwrite error, when it exists in the code, it will always cause a memory-overwritten condition. However, from a black-box testing perspective, we will never have a chance to see the symptom of this error until the specific overwritten byte(s) of code or data is executed or read. In this example, the set of steps represents the exact set of black-box activities. The memory-overwrite error represents the actual error in the code. The condition in which the overwritten byte is executed or read represents the dynamic operating environment or condition needed to reveal (reproduce) the error."

Write Good Change Requests



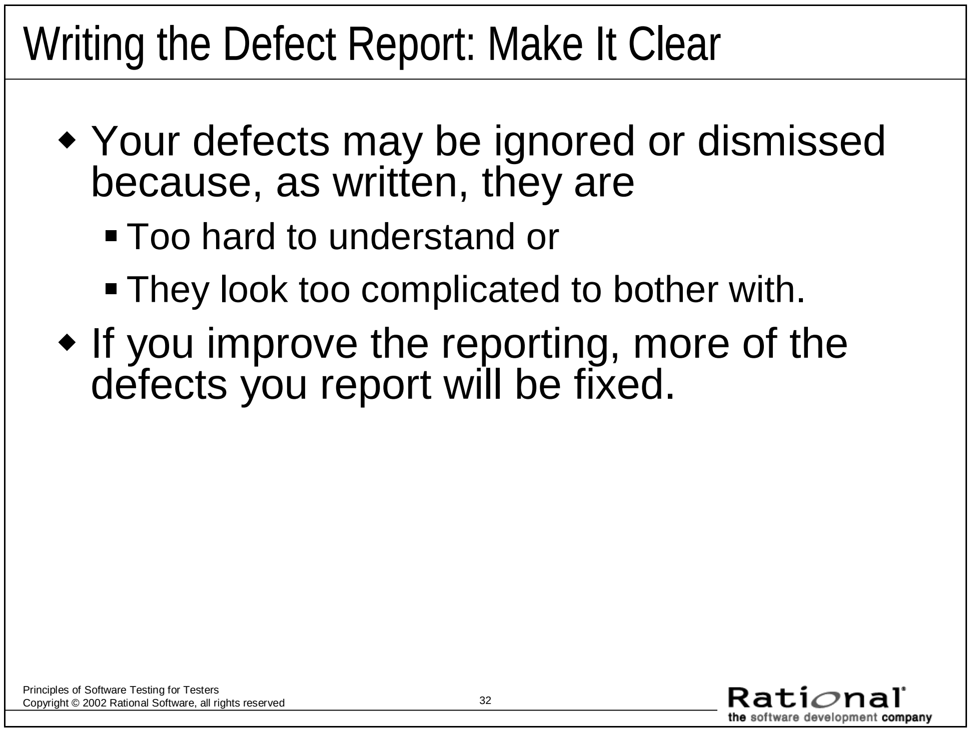
Module 6 Agenda

- Advocate repairing the important problems
- Investigate problems effectively
- **Write good change requests**

Principles of Software Testing for Testers

31

Writing the Defect Report: Make It Clear



Writing the Defect Report: Make It Clear

- Your defects may be ignored or dismissed because, as written, they are
  - Too hard to understand or
  - They look too complicated to bother with.
- If you improve the reporting, more of the defects you report will be fixed.

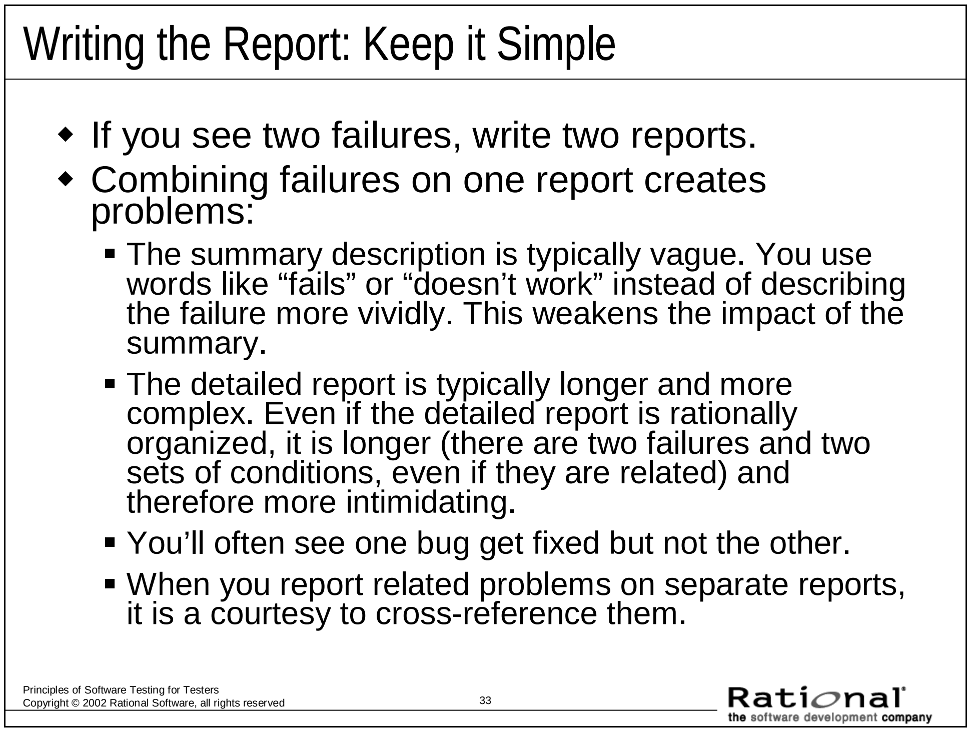
Cem Kaner writes:

Some testing groups train their staff to make a bug reproducible and stop there. They don't do much follow-up testing and they don't polish their reports.

I've had the opportunity to review the bug data of several companies, including a great deal of data from one company that is well respected for the quality of its code and the quality-consciousness of its engineers, and data from several projects that I've managed with different development teams.

My conclusion is that even very quality-conscious engineering teams are more likely to take a problem seriously, investigate it and (if appropriate) fix it if the report is clear and well-written.

Writing the Report: Keep it Simple



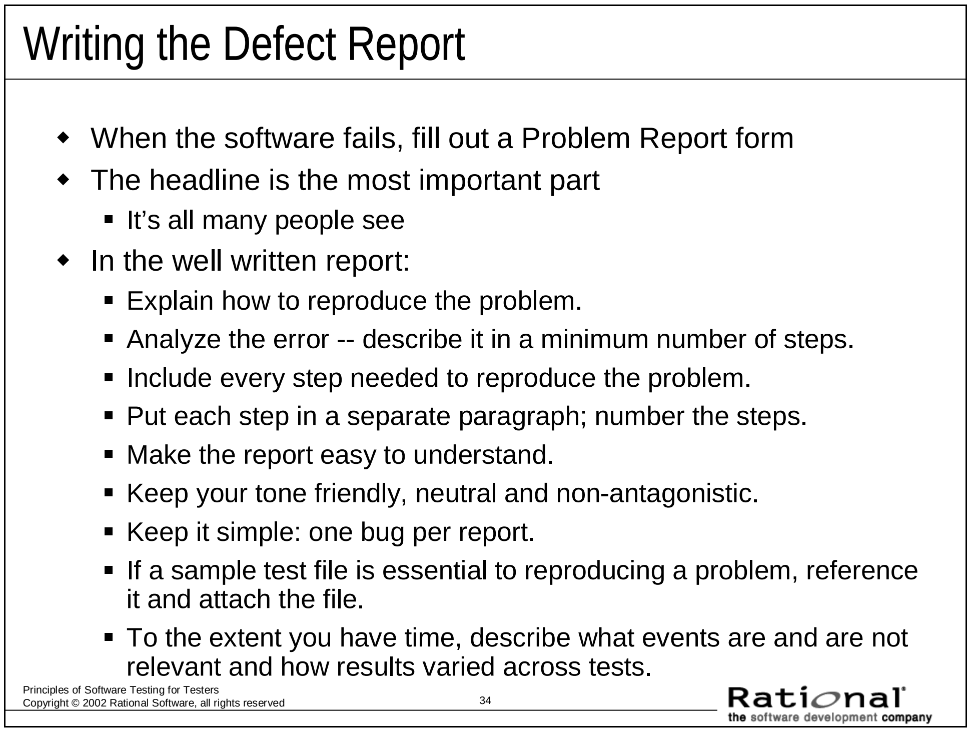
Writing the Report: Keep it Simple

- If you see two failures, write two reports.
- Combining failures on one report creates problems:
  - The summary description is typically vague. You use 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 longer and more complex. 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.

Principles of Software Testing for Testers

33
Rational the software development company

Writing the Defect Report



Writing the Defect Report

- When the software fails, fill out a Problem Report form
- The headline is the most important part
  - It's all many people see
- In the well written report:
  - Explain how to reproduce the problem.
  - Analyze the error -- describe it in a minimum number of steps.
  - Include every step needed to reproduce the problem.
  - Put each step in a separate paragraph; number the steps.
  - Make the report easy to understand.
  - Keep your tone friendly, 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 file.
  - To the extent you have time, describe what events are and are not relevant and how results varied across tests.

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 34

Make sure that you **always** file a report. Don't just walk over the developer cubicle and verbally communicate, and then skip writing the report. You will not be able to track it.

Emotional tone of defect reports is very important. Reports that insult or demean the developer will hurt your credibility more than the developer's, and likely won't be enough motivation to fix the problem.

Use headlines effectively

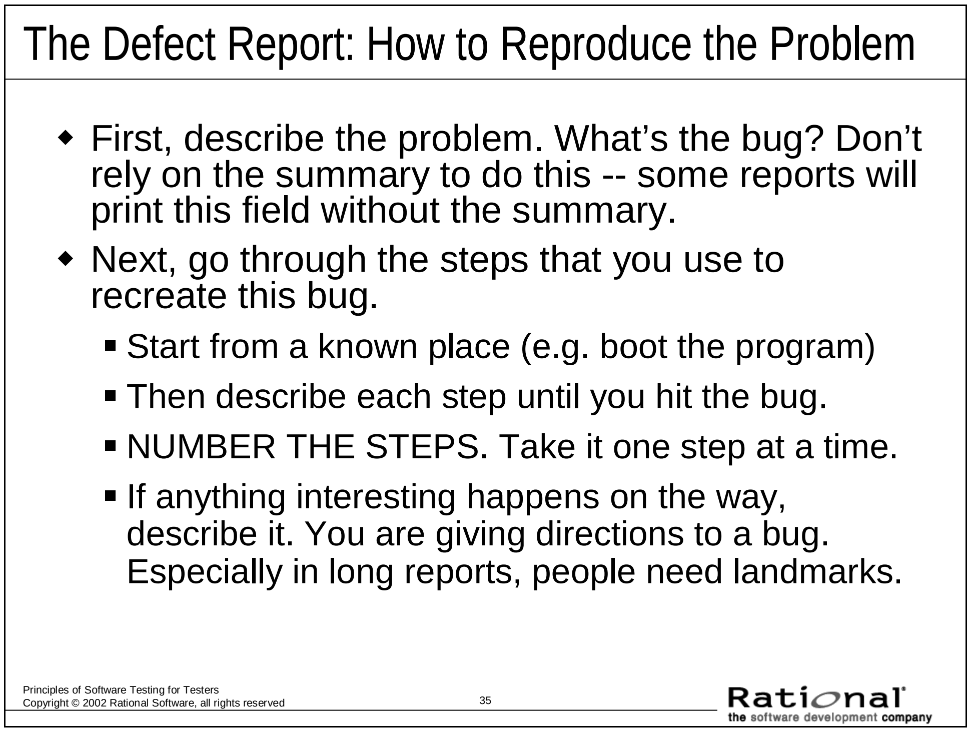
This one-line description of the problem is the most important part of the report.

- 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 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 Defect Report: How to Reproduce the Problem



The Defect Report: How to Reproduce the Problem

- 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)
  - 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 directions to a bug. Especially in long reports, people need landmarks.

Principles of Software Testing for Testers
35

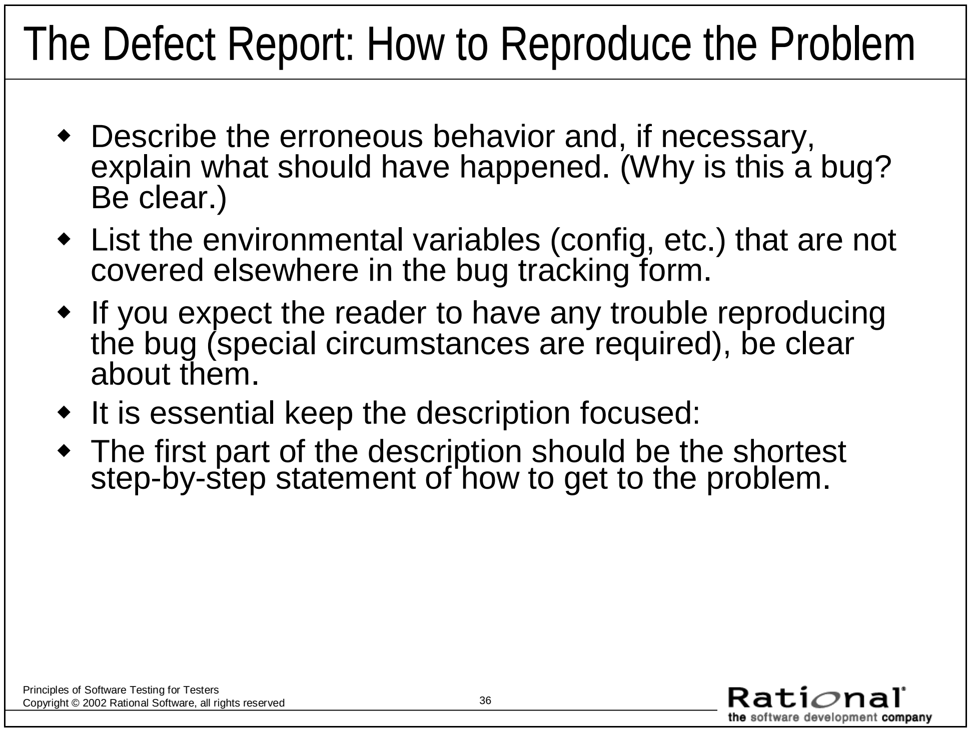
The following policy is not uncommon:

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

Many bug tracking systems include a field asking whether you can reproduce the problem. Even if your system doesn't include this, 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 Defect Report: How to Reproduce the Problem



The Defect Report: How to Reproduce the Problem

- 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.
- 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.

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 36

What if 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 output if you make the file a few bytes longer

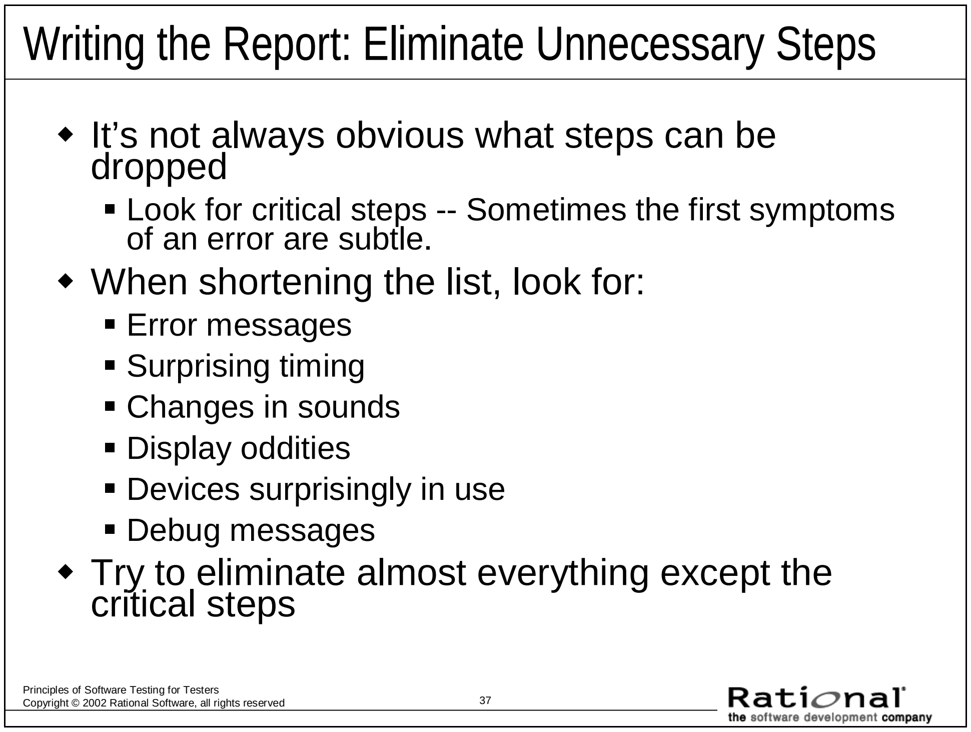
This is all useful information for the developer and you should include it. But to make the report clear:

- Start the report with a step-by-step description of the shortest series of steps needed to produce the failure.
- **Then add a section that says "NOTES"** and describe, one by one, in this section the additional variations and the effect on the observed failure.

Additional notes might include comments that:

- The bug won't show up if you do step X between step Y and step Z.
- Explain your reasoning for running this test.
- Explain why you think this is an interesting bug.

Writing the Report: Eliminate Unnecessary Steps



Writing the Report: Eliminate Unnecessary Steps

- It's not always obvious what steps can be dropped
  - Look for critical steps -- Sometimes the first symptoms of an error are subtle.
- When shortening the list, look for:
  - Error messages
  - Surprising timing
  - Changes in sounds
  - Display oddities
  - Devices surprisingly in use
  - Debug messages
- Try to eliminate almost everything except the critical steps

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 37

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.

Sometimes the first indicator that the system is working differently is that it sounds a little different than normal.

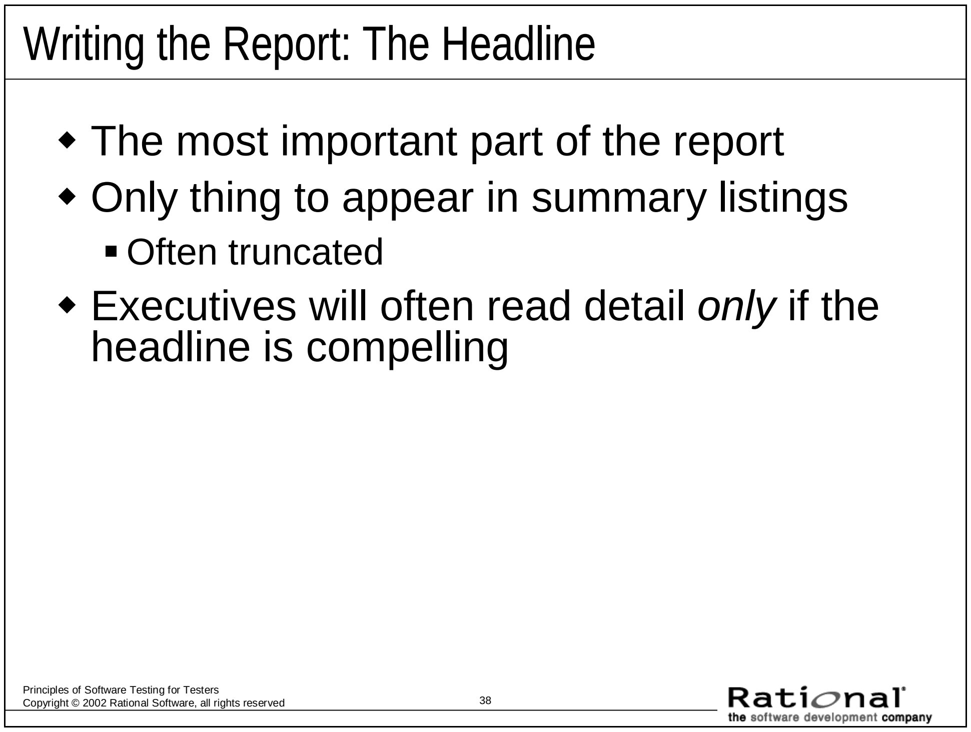
Display oddities, such as a flash, a repainted screen, a cursor that jumps back and forth, multiple cursors, misaligned text, slightly distorted graphics, doubled characters, omitted characters, or display droppings (pixels that are still colored even though the character or graphic that contained them was erased or moved).

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 defect 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.

Writing the Report: The Headline



Writing the Report: The Headline

- The most important part of the report
- Only thing to appear in summary listings
  - Often truncated
- Executives will often read detail *only* if the headline is compelling

Principles of Software Testing for Testers

38
Rational the software development company

Use headlines effectively

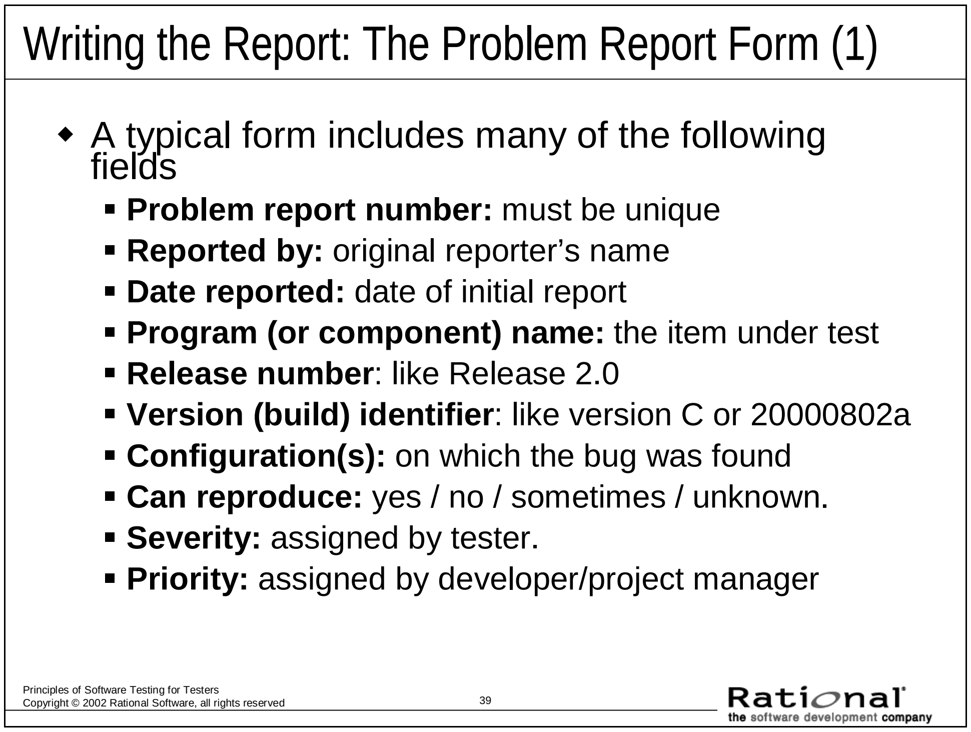
This one-line description of the problem is the most important part of the report.

- 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.)

Writing the Report: The Problem Report Form (1)



Writing the Report: The Problem Report Form (1)

- A typical form includes many of the following fields
  - **Problem report number:** must be unique
  - **Reported by:** original reporter’s name
  - **Date reported:** date of initial report
  - **Program (or component) name:** the item under test
  - **Release number**: like Release 2.0
  - **Version (build) identifier**: like version C or 20000802a
  - **Configuration(s):** on which the bug was found
  - **Can reproduce:** yes / no / sometimes / unknown.
  - **Severity:** assigned by tester.
  - **Priority:** assigned by developer/project manager

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 39

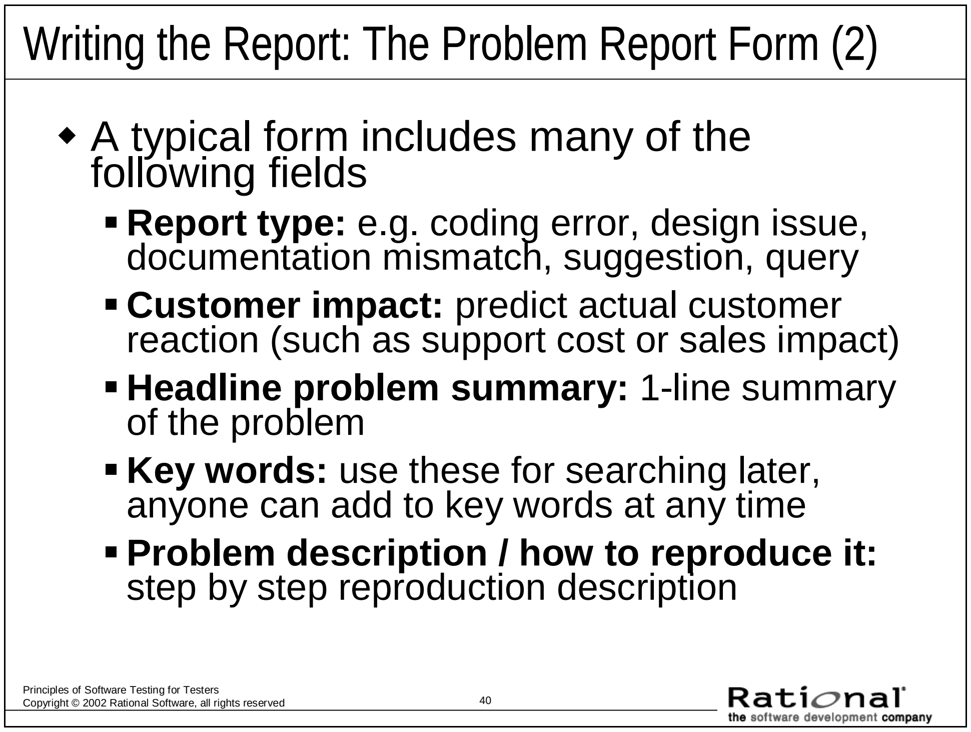
VERSION: Make sure you get this right or the whole report may be wasted.

CONFIGURATIONS: List the hardware and software configuration(s) under which the bug was found and replicated. Some of the better-designed databases allow testers to list “standard” configurations (a list of characteristics of their most commonly used machines). The tester then refers to the configurations by name, rather than describing the same system in detail time after time. Additionally, some databases allow the tester to list two or more configurations for the same bug.

CAN REPRODUCE: (Unknown can arise, for example, when the repro configuration is at a customer site and not available to the lab.)

PRIORITY vs. SEVERITY. The priority of the bug determines when it should be fixed. A bug that blocks further testing or development might be fixed sooner than one that crashes the system and corrupts data. The project manager (and not the tester) is usually the one to determine the priority of the bug. Normally, the tester decides how severe it is.

Writing the Report: The Problem Report Form (2)



Writing the Report: The Problem Report Form (2)

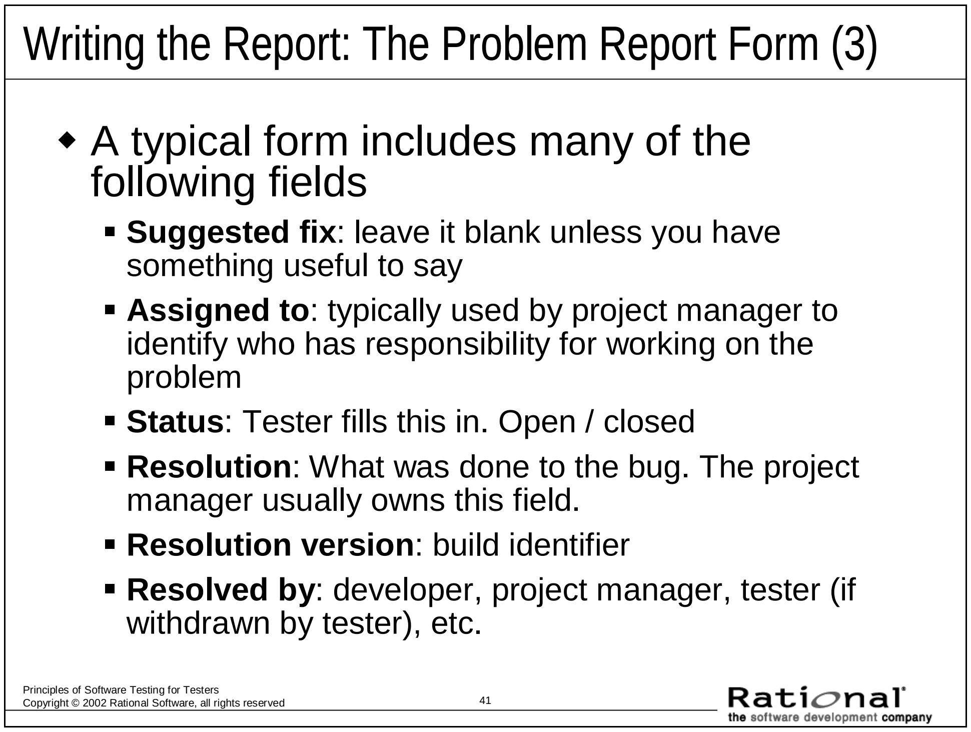
- A typical form includes many of the following fields
  - **Report type:** e.g. coding error, design issue, documentation mismatch, suggestion, query
  - **Customer impact:** predict actual customer reaction (such as support cost or sales impact)
  - **Headline problem summary:** 1-line summary of the problem
  - **Key words:** use these for searching later, anyone can add to key words at any time
  - **Problem description / how to reproduce it:** step by step reproduction description

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

- Coding Error: *The program doesn't do what the developer would expect it to do.*
- Design Issue: *It's doing what the developer 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 developer (via a defect 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.*

CUSTOMER IMPACT is often left blank. When used, it is typically filled in by tech support or someone else who can credibly estimate customer impact.

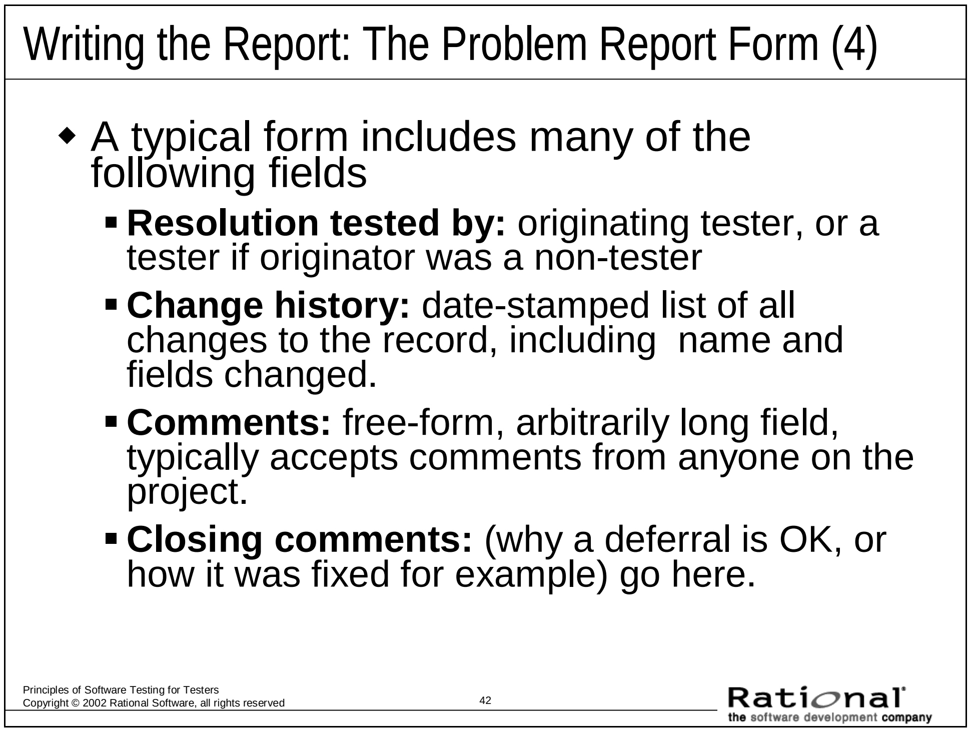
Writing the Report: The Problem Report Form (3)



Writing the Report: The Problem Report Form (3)

- A typical form includes many of the following fields
  - **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 working on the problem
  - **Status**: Tester fills this in. Open / closed
  - **Resolution**: What was done to the bug. The project manager usually owns this field.
  - **Resolution version**: build identifier
  - **Resolved by**: developer, project manager, tester (if withdrawn by tester), etc.

Writing the Report: The Problem Report Form (4)



Writing the Report: The Problem Report Form (4)

- A typical form includes many of the following fields
  - **Resolution tested by:** originating tester, or a tester if originator was a non-tester
  - **Change history:** date-stamped list of all changes to the record, including name and fields changed.
  - **Comments:** free-form, arbitrarily long field, typically accepts comments from anyone on the project.
  - **Closing comments:** (why a deferral is OK, or how it was fixed for example) go here.

Rational the software development company

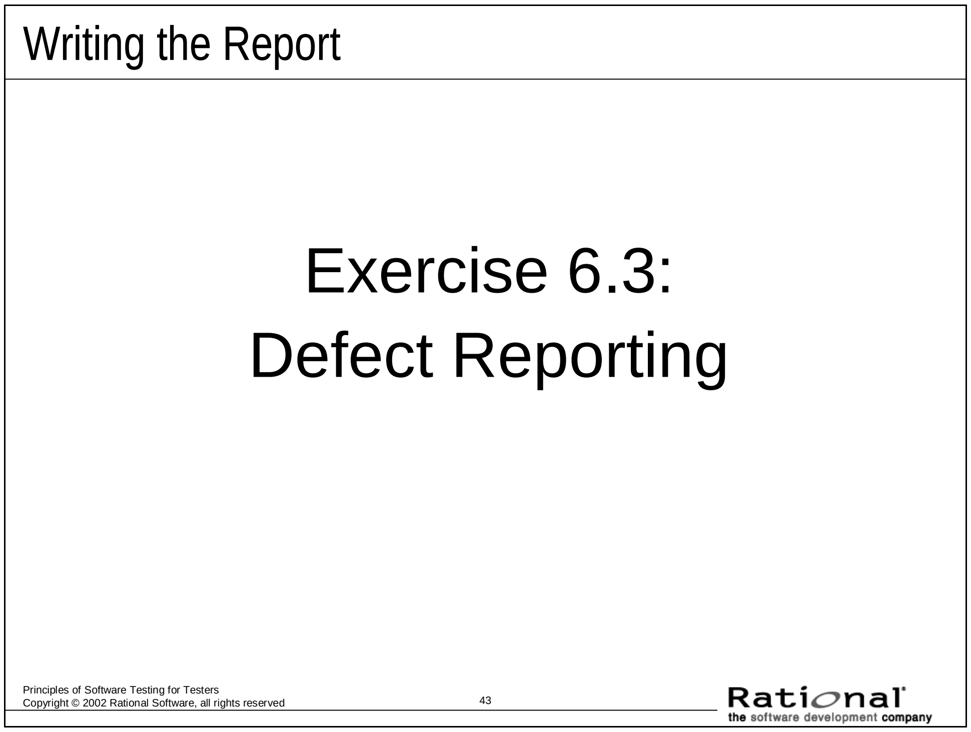
RESOLUTION: Common resolutions include:

- Pending: the bug is still being worked on.
- Fixed: the developer says it's fixed. Now check it.
- Cannot reproduce: When the developer can't make the failure happen, you add details, reset the resolution to Pending, and notify the developer.
- Deferred: It's a bug, but we'll fix it later.
- As Designed: The program works as it's supposed to.
- Need Info: The developer needs more info from you. She has probably asked a question in the comments.
- Duplicate: This is just a repeat of another defect 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.

COMMENTS: Testers developers, tech support (in some companies) and others may all add comments to this field. In many companies, this field is an ongoing discussion of repro conditions, etc., until the bug is resolved.

CLOSING COMMENTS. This field is valuable for recording progress with difficult or controversial bugs. Write carefully. Just like e-mail, comments are easy to misunderstand.

Writing the Report



Writing the Report

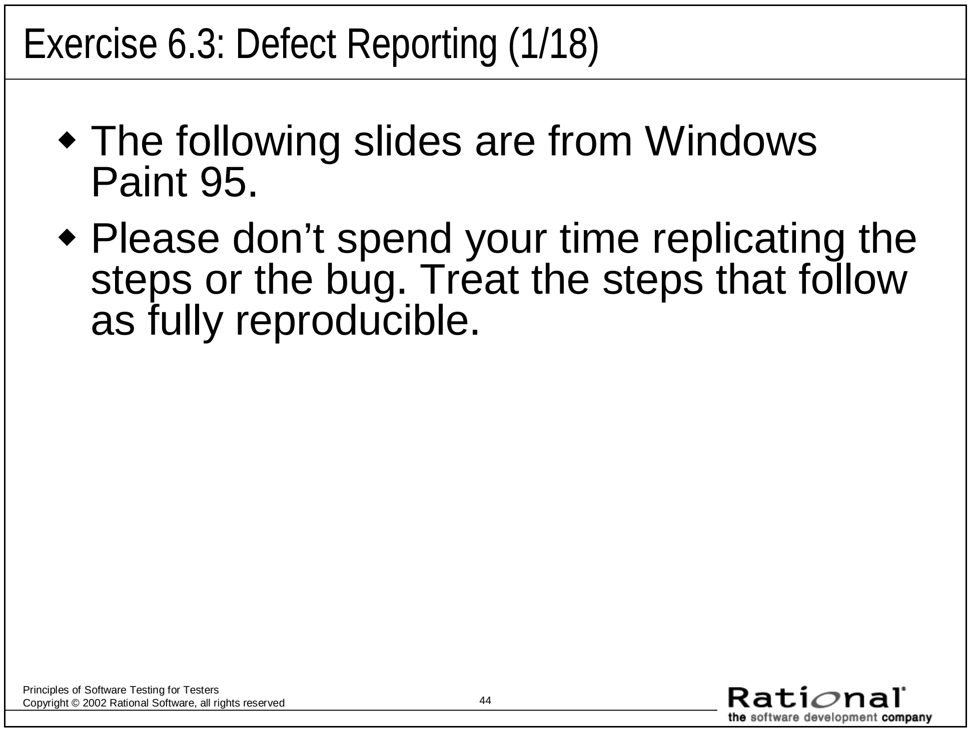
Exercise 6.3:
Defect Reporting

Principles of Software Testing for Testers

43

Rational
the software development company

Exercise 6.3: Defect Reporting (1/18)



Exercise 6.3: Defect Reporting (1/18)

- The following slides are from Windows Paint 95.
- Please don't spend your time replicating the steps or the bug. Treat the steps that follow as fully reproducible.

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.

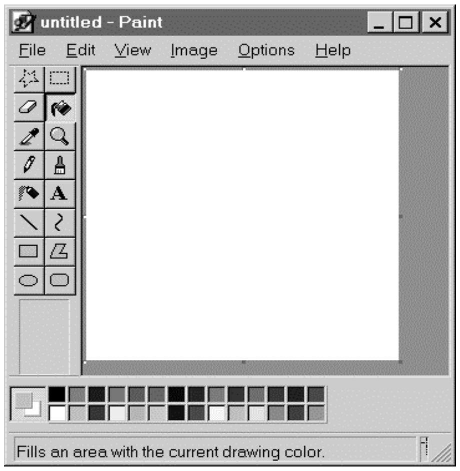
In a paint program, you cannot select the circle once you've drawn it. You can select an area that includes the dots that make up the circle, but that area is simply a bitmap and none of the dots in it have any relationship to any of the others.

Exercise 6.3: Defect Reporting (2/18)

Exercise 6.3: Defect Reporting (2/18)

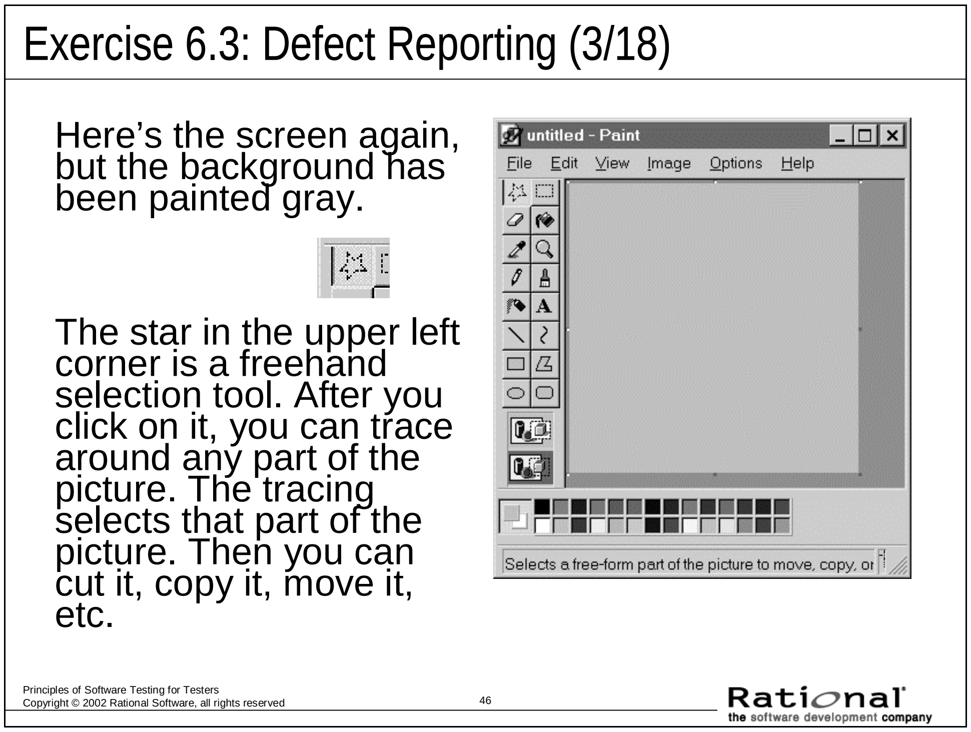
Here's the opening screen. The background is white. The first thing that we'll do is select the Paint Can

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.



Fills an area with the current drawing color.

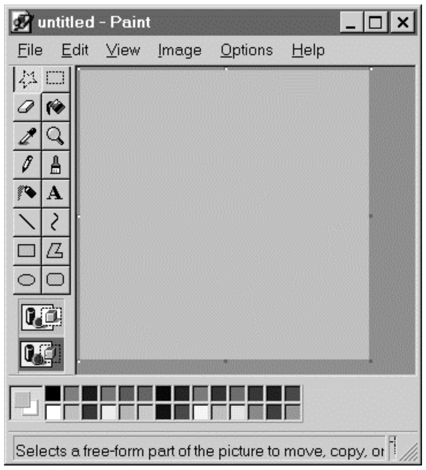
Exercise 6.3: Defect Reporting (3/18)



Exercise 6.3: Defect Reporting (3/18)

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.



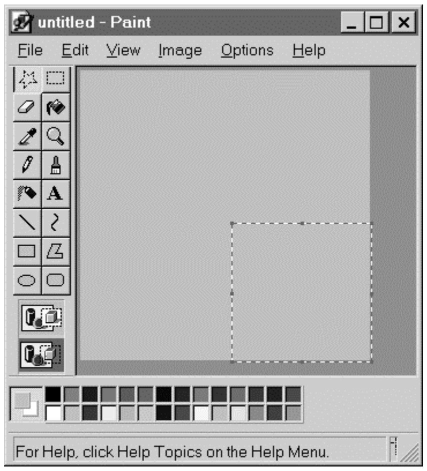
Selects a free-form part of the picture to move, copy, or

Exercise 6.3: Defect Reporting (4/18)

Exercise 6.3: Defect Reporting (4/18)

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.



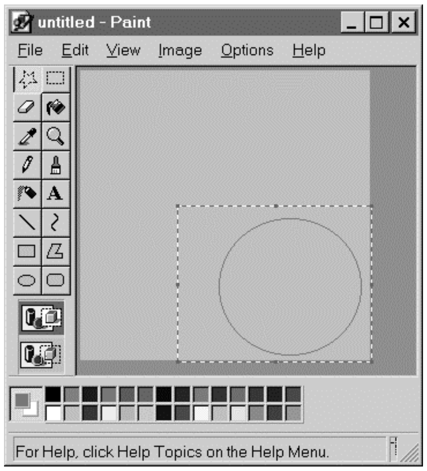
For Help, click Help Topics on the Help Menu.

Exercise 6.3: Defect Reporting (5/18)

Exercise 6.3: Defect Reporting (5/18)

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.

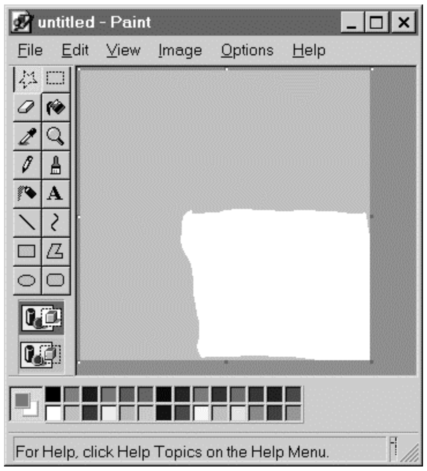


Exercise 6.3: Defect Reporting (6/18)

Exercise 6.3: Defect Reporting (6/18)

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

The jagged border shows exactly the area that was selected.

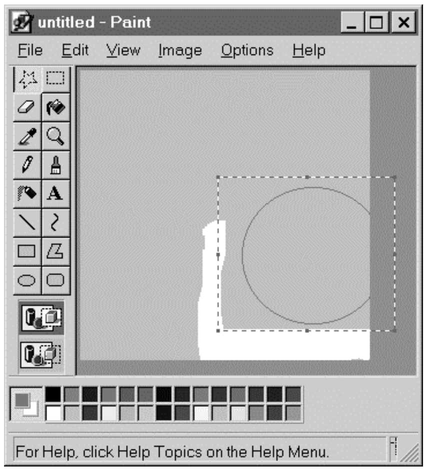

untitled - Paint
File Edit View Image Options Help
For Help, click Help Topics on the Help Menu.

Exercise 6.3: Defect Reporting (7/18)

Exercise 6.3: Defect Reporting (7/18)

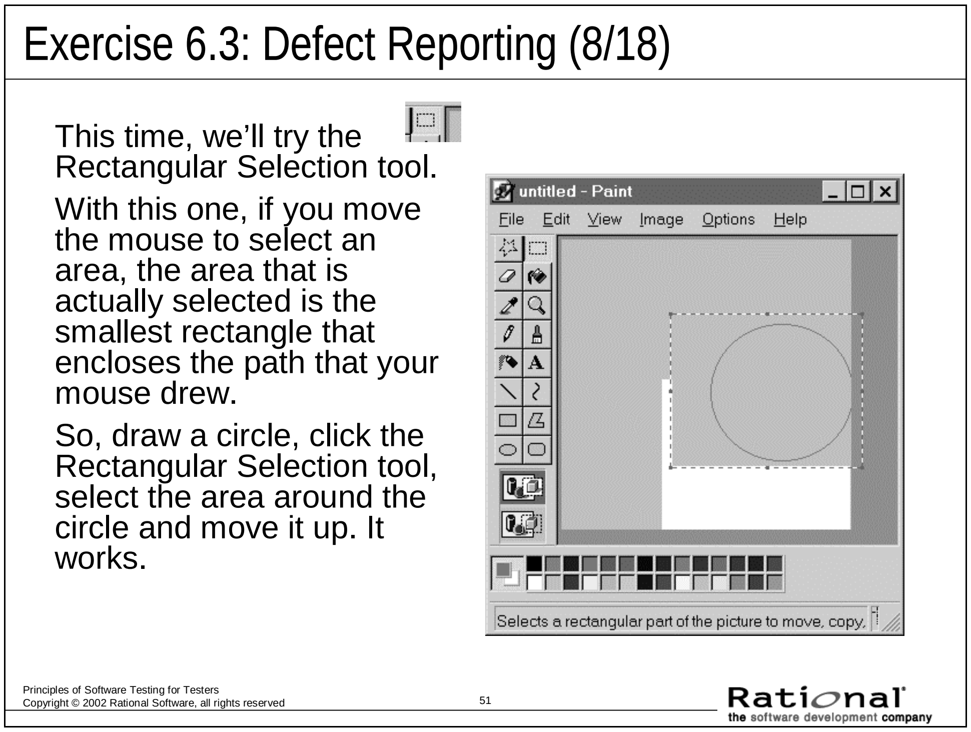
Next, select the area around the circle and drag it up and to the right.

This works.



For Help, click Help Topics on the Help Menu.

Exercise 6.3: Defect Reporting (8/18)



Exercise 6.3: Defect Reporting (8/18)

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.

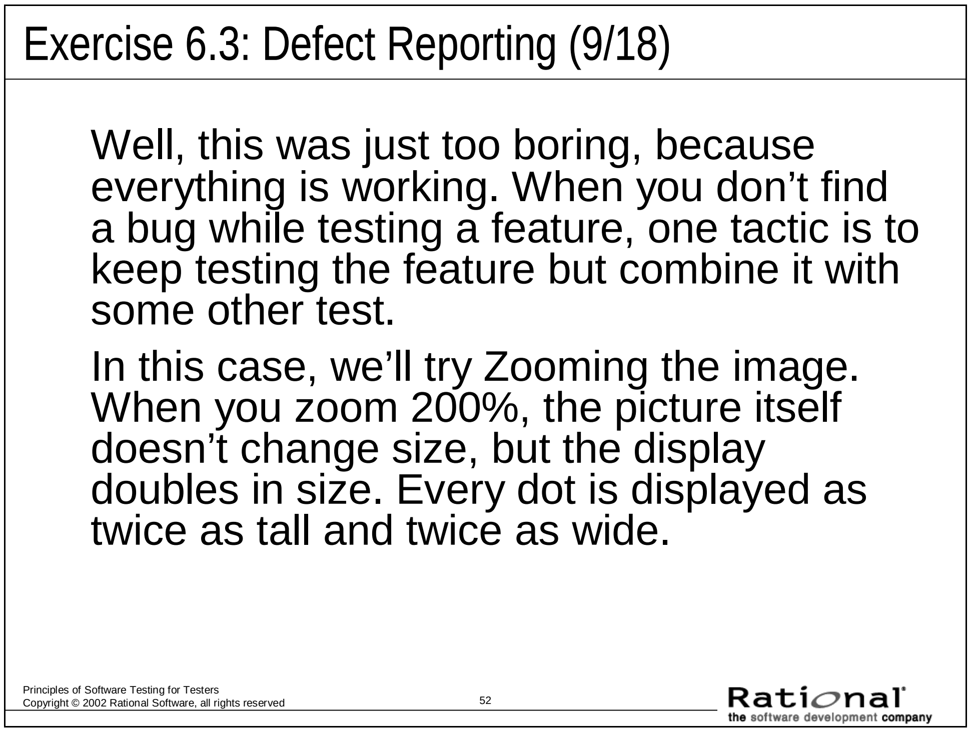
So, draw a circle, click the Rectangular Selection tool, select the area around the circle and move it up. It works.

Selects a rectangular part of the picture to move, copy,

51

Rational the software development company

Exercise 6.3: Defect Reporting (9/18)



Exercise 6.3: Defect Reporting (9/18)

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.

Principles of Software Testing for Testers

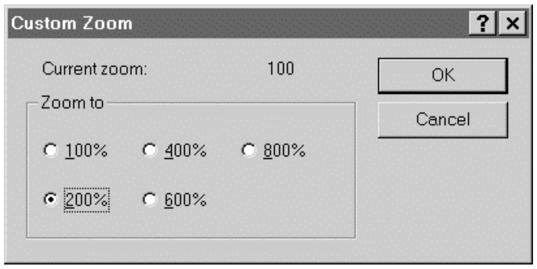
52

Rational
the software development company

Exercise 6.3: Defect Reporting (10/18)

Exercise 6.3: Defect Reporting (10/18)

Bring up the Custom Zoom dialog, and select 200% zoom, click OK.


Custom Zoom

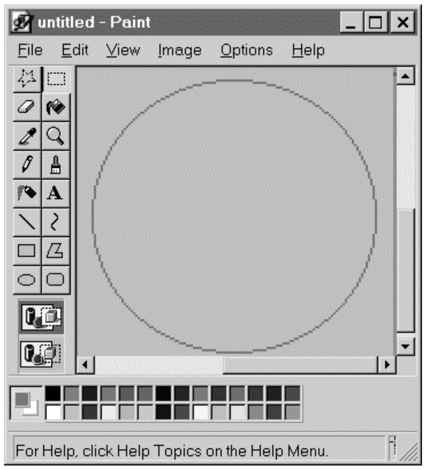
Current zoom: 100

Zoom to: ( ) 100% ( ) 400% ( ) 800% [✓ 200%] ( ) 600%

Exercise 6.3: Defect Reporting (11/18)

Exercise 6.3: Defect Reporting (11/18)

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.

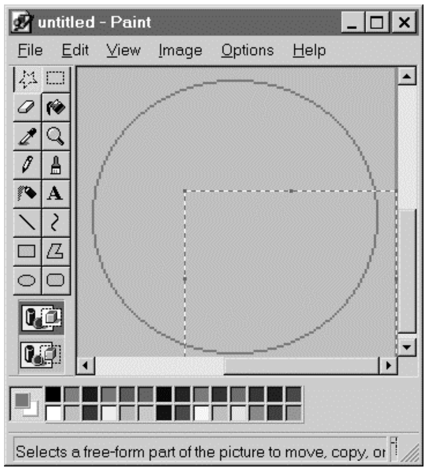


For Help, click Help Topics on the Help Menu.

Exercise 6.3: Defect Reporting (12/18)

Exercise 6.3: Defect Reporting (12/18)

- 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.

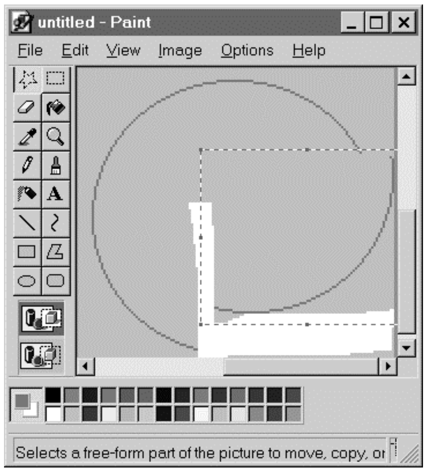


Selects a free-form part of the picture to move, copy, or

Exercise 6.3: Defect Reporting (13/18)

Exercise 6.3: Defect Reporting (13/18)

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

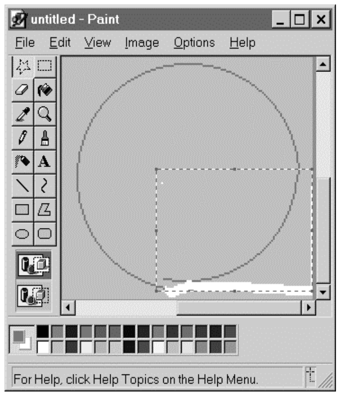


Selects a free-form part of the picture to move, copy, or

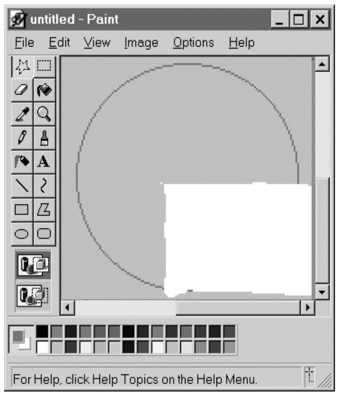
Exercise 6.3: Defect Reporting (14/18)

Exercise 6.3: Defect Reporting (14/18)

- 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.



For Help, click Help Topics on the Help Menu.



For Help, click Help Topics on the Help Menu.

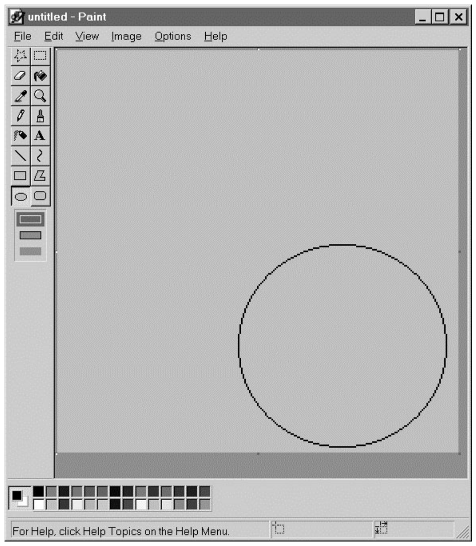
57

Rational
the software development company

Exercise 6.3: Defect Reporting (15/18)

Exercise 6.3: Defect Reporting (15/18)

- Draw the circle, zoom to 200%, and this time, grow the window so you can see the whole drawing area.


untitled - Paint
File Edit View Image Options Help
For Help, click Help Topics on the Help Menu.

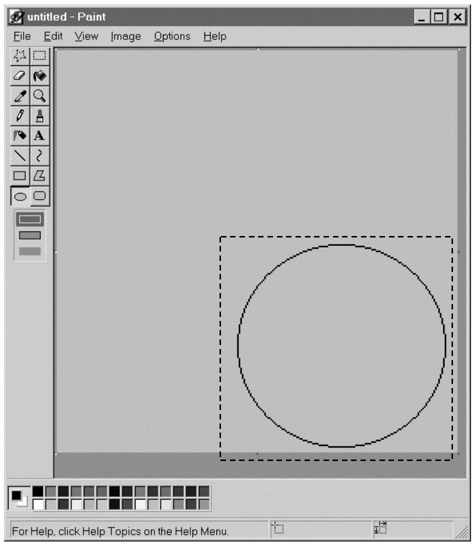
58

Rational
the software development company

Exercise 6.3: Defect Reporting (16/18)

Exercise 6.3: Defect Reporting (16/18)

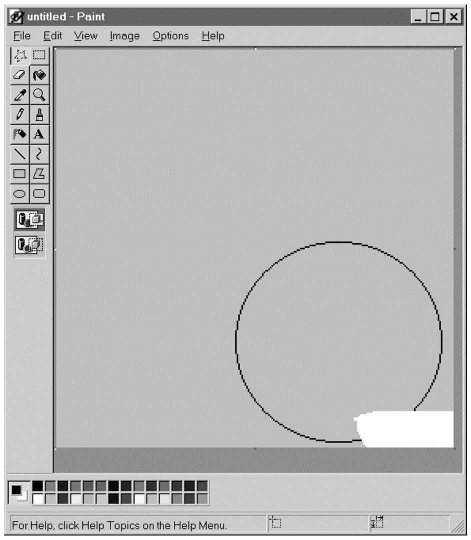
- Now, select the circle. That seems to work.



Exercise 6.3: Defect Reporting (17/18)

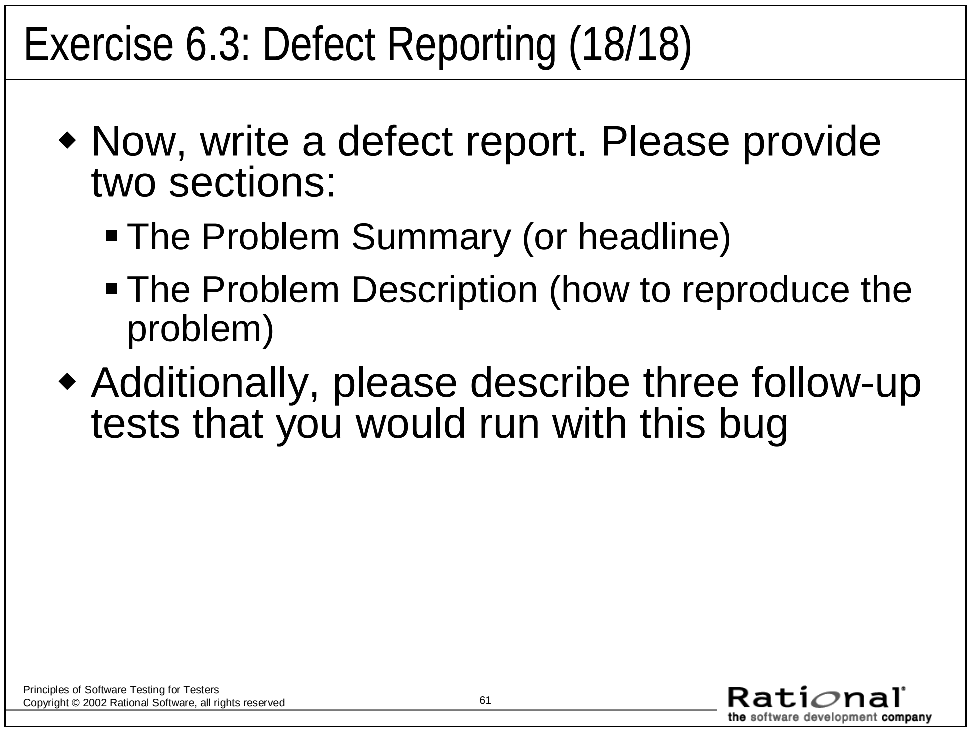
Exercise 6.3: Defect Reporting (17/18)

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



For Help, click Help Topics on the Help Menu.

Exercise 6.3: Defect Reporting (18/18)

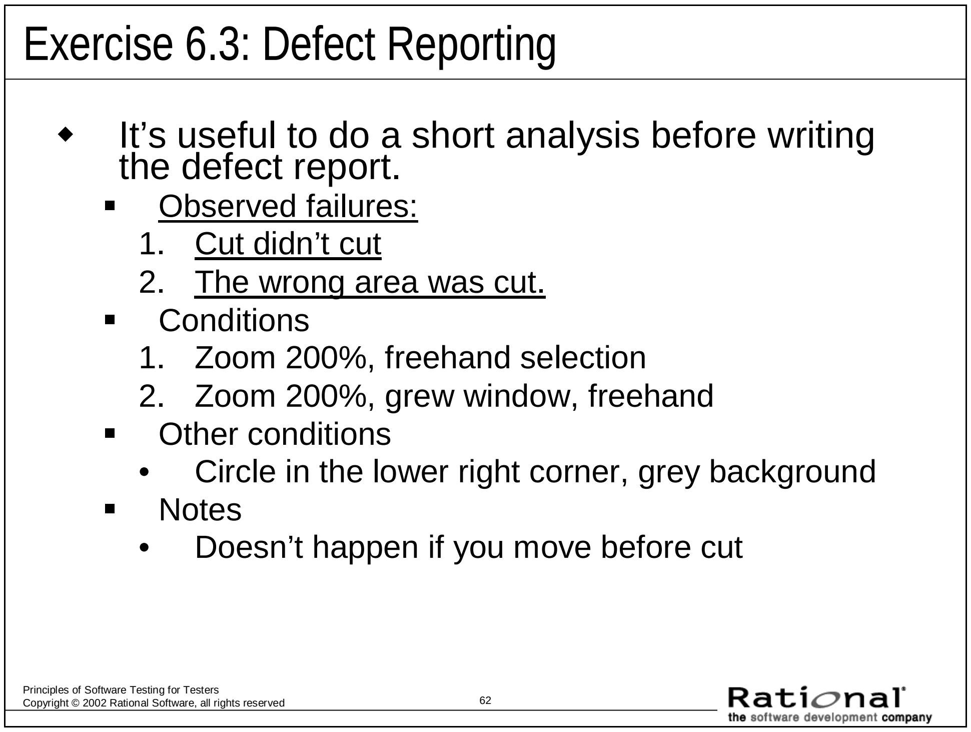


Exercise 6.3: Defect Reporting (18/18)

- Now, write a defect report. Please provide two sections:
  - The Problem Summary (or headline)
  - The Problem Description (how to reproduce the problem)
- Additionally, please describe three follow-up tests that you would run with this bug

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 61

Exercise 6.3: Defect Reporting



Exercise 6.3: Defect Reporting

- It’s useful to do a short analysis before writing the defect report.
  - Observed failures:
    1. Cut didn’t cut
    2. The wrong area was cut.
  - Conditions
    1. Zoom 200%, freehand selection
    2. Zoom 200%, grew window, freehand
  - Other conditions
    - Circle in the lower right corner, grey background
  - Notes
    - Doesn’t happen if you move before cut

62

Rational the software development company

Here, there are two different failures. Normally, if you see different failures, you write different defect reports. Here, these look related, so you would cross-reference them.

When people merge both of these bugs onto one problem report form, their summary is generally much longer and more vague.

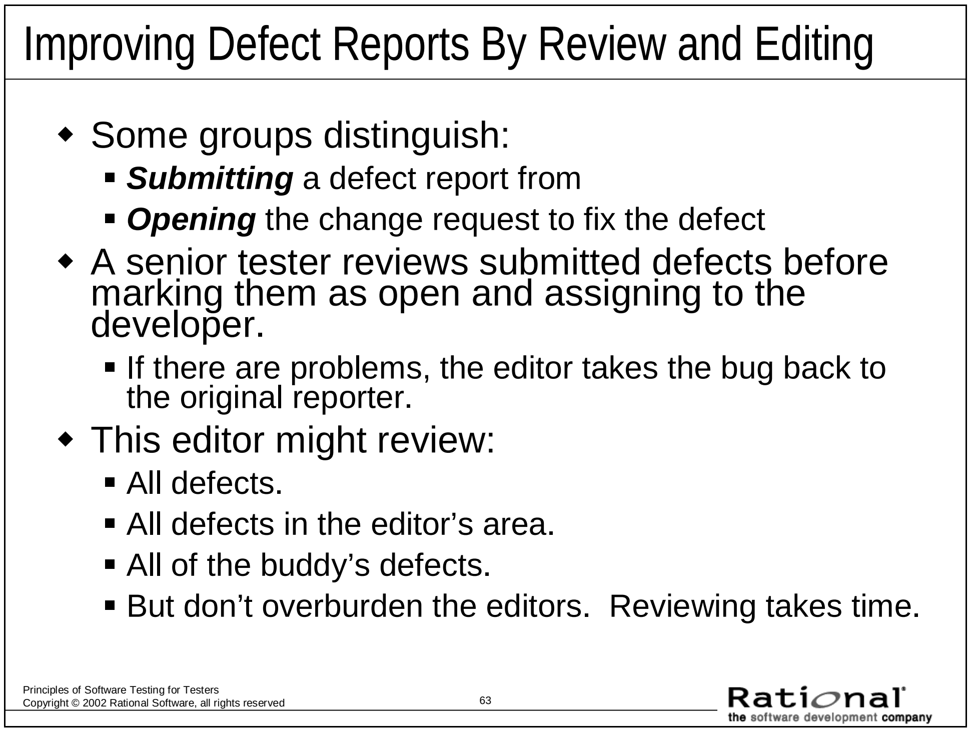
Here are typical (good) summaries when the bugs are reported separately:

1. Cut doesn’t cut (interacts with zoom, freehand)
2. Cut the wrong area (interacts with zoom, freehand, grow window)

Here are typical summaries when the bugs are reported together (provided here as examples of what not to do):

1. Problem with cutting
2. Cut doesn’t work correctly

Improving Defect Reports By Review and Editing



Improving Defect Reports By Review and Editing

- Some groups distinguish:
  - ***Submitting*** a defect report from
  - ***Opening*** the change request to fix the defect
- A senior tester reviews submitted defects before marking them as open and assigning to the developer.
  - If there are problems, the editor takes the bug back to the original reporter.
- This editor might review:
  - All defects.
  - All defects in the editor's area.
  - All of the buddy's defects.
  - But don't overburden the editors. Reviewing takes time.

Principles of Software Testing for Testers
Copyright © 2002 Rational Software, all rights reserved 63

Rational the software development company

The tester reviewing submitted defects:

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

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

If the reporter was outside the test group, the reviewer simply checks basic facts with him.

If the reporter was a tester, the reviewer points out problems in order to further the tester's training.

This tester might review:

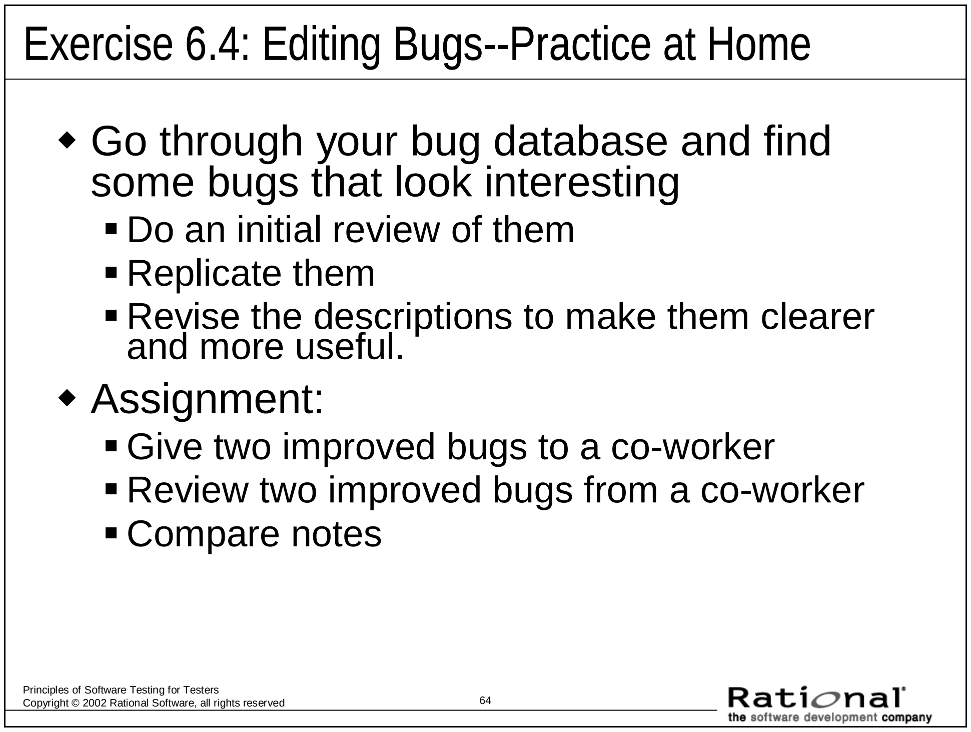
- all defects
- all defects in the editor's area
- all of the buddy's defects.

Beware of overburdening the reviewing testers.

The reviewer will go through a learning curve (learning about parts of the system or types of tests that haven't been 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 it is read.

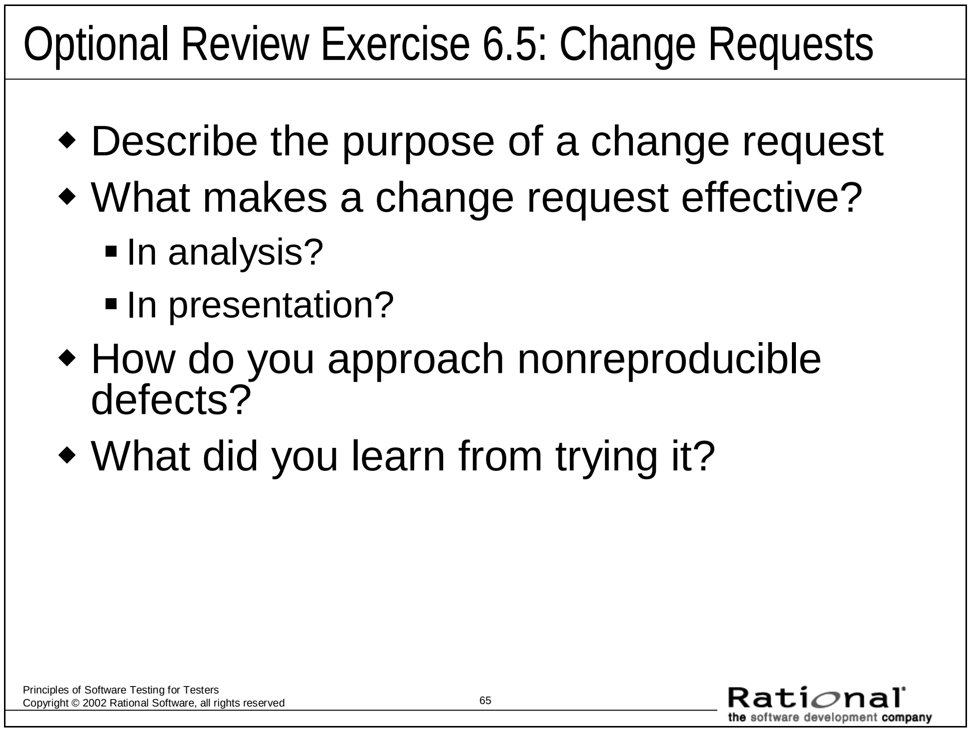
Exercise 6.4: Editing Bugs--Practice at Home



Exercise 6.4: 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

Optional Review Exercise 6.5: Change Requests



Optional Review Exercise 6.5: Change Requests

- Describe the purpose of a change request
- What makes a change request effective?
  - In analysis?
  - In presentation?
- How do you approach nonreproducible defects?
- What did you learn from trying it?

Principles of Software Testing for Testers

65