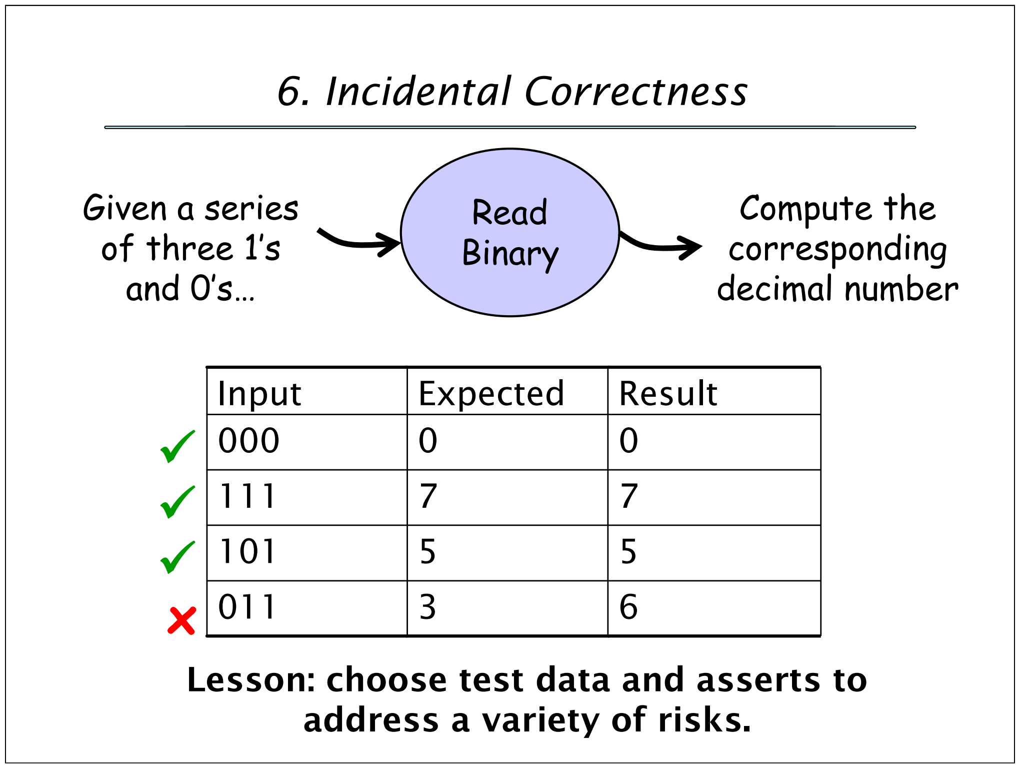
You're Kidding. It Does WHAT in Production!?!

Common reasons production software still has bugs even when written by the most diligent and disciplined XP teams, and what you can do about it.

Elisabeth Hendrickson
Quality Tree Software, Inc.
esh@qualitytree.com

v 1.0. Copyright (c) 2007 Quality Tree Software, Inc.

 Copyright (c) 2007 Quality Tree Software, Inc.



Right for the Wrong Reason

It's easy to have tests that fail to fail. Common causes include:

- Missing cases
- Missing asserts
- Weak asserts
- Weak/limited/artificially simple test data

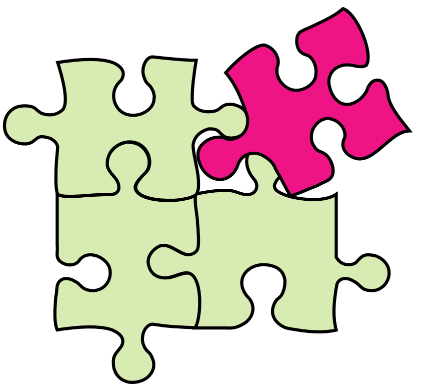
Copyright (c) 2007 Quality Tree Software, Inc.

5. External Dependencies

You built the software using libraries or external services you don't control.

The external dependency changed, and your software is now busted.

Oops.



Lesson: Don't Test Other People's Code…
But DO Test Your Expectations of It.

Tests Aren't Just for Testing Your Code…

They're also for testing your assumptions.

If your software relies on external dependencies, whether a 3rd party or open source library, or an external Web service, or even the operating system, it's a good idea to create tests that make assertions about your assumptions. Such tests serve multiple purposes:

1. They provide living, technical documentation about the dependency.

2. They give you an early warning system if the external dependency changes.

3. They provide a fast and easy way to sanity check the effect of upgrading/downgrading the dependency.

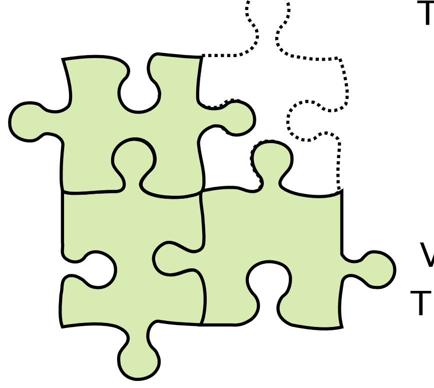
Mitigating the Risk of Mocking

When you mock external dependencies to simplify your tests, there's always a risk that the tests will be green and the software will fall over dead in production. (This risk is commonly cited as a reason not to mock. I think it's a reason to mock carefully.)

To ensure your mocks aren't hiding failure conditions, create tests for each mocked aspect of the external dependency to assert the response from the mock matches the response from the external dependency.

v 1.0. Copyright (c) 2007 Quality Tree Software, Inc.

4. Production Server Configuration Differences



The software uses something in the development environment: Exes, Gems, Jars, Directories, Files, Databases, Data, Registry Settings, Environment Variables, the hostname, Etc.

The production server doesn't have it.

Oops again.

Lessons: Practice deploying early and often. Write tests to verify the integrity of the production environment. Automate deployment.

Deployment Configuration Variations

No matter how much testing you do, testing can only tell you so much about how the software will behave in the wild.

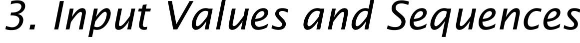
Programmers sometimes say: "All the problems we're having in production are just configuration issues. The code is solid."

This statement misses the point: a system is more than just code. The success of the system will be judged by how well the code works in the production environment.

Deploy early, deploy often, and automate checks of the production system.

v 1.0. Copyright (c) 2007 Quality Tree Software, Inc.

3. Input Values and Sequences



You assume they're reasonable people who will do reasonable things.

They aren't and they won't.

Lesson: Vary anything that can be varied.

Users: Perpetually Surprising

Users do the most amazing things. They try to log on in two different browser windows. They ignore error messages, and bury error dialogs at the bottom of the screen to avoid looking at them. They enter invalid data. They do things out of order. They bookmark pages that shouldn't be bookmarked. They edit files that shouldn't be edited. They change settings that shouldn't be changed. They print pages not designed to be printed. They make invalid assumptions about what the software should and should not do.

Users do all these things because they just want to use the software, not understand it.

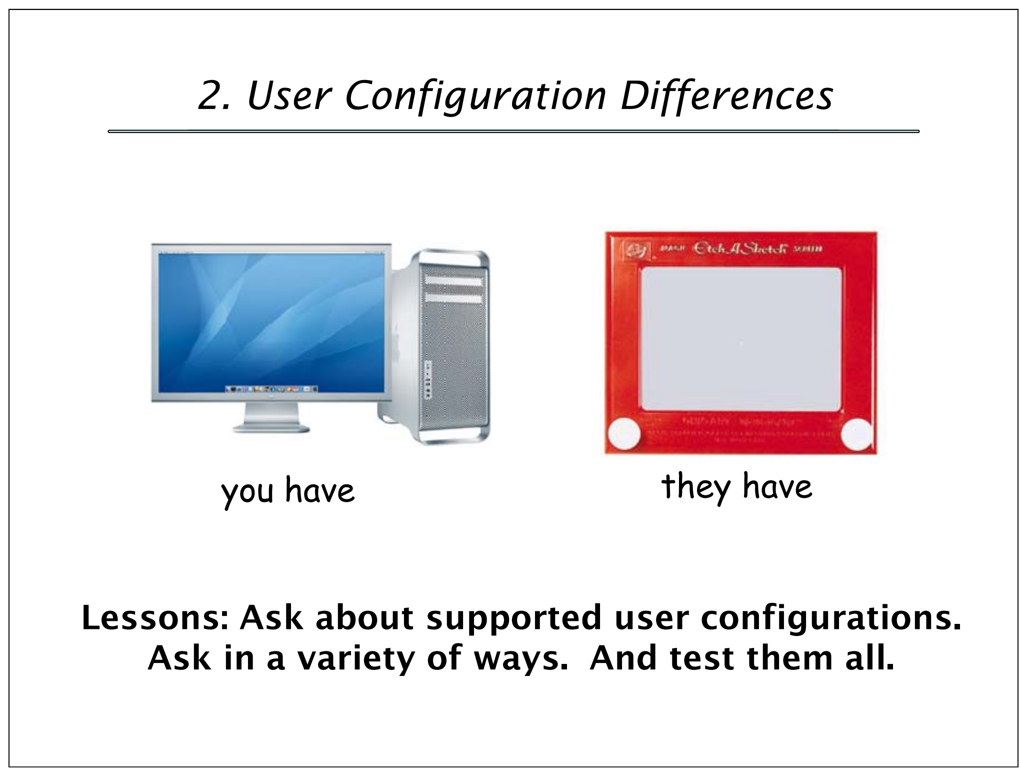
Hackers: Perpetually Devious

Hackers, on the other hand, are trying to thwart the system.

Some are truly malicious, trying to find ways to access protected data such as social security numbers or credit card information. These people not only seek to understand the software, but also all the built in protections.

Hackers build up arsenals of sneaky, dirty tricks. Tricks like entering <script> commands in text fields on websites or injecting SQL or exploiting buffer overrun vulnerabilities to compromise a system.

 Copyright (c) 2007 Quality Tree Software, Inc.



You Probably Have a Better System than Your Users

You fought hard to get your Mac Pro 3.0GHz Quad-Core Intel Xeon 5300 series, 8-core processing system with 16Gb memory and dual 30-inch displays.

The users have 5-year-old consumer PCs with 128Mb memory, running IE 5.

Oddly enough, the software behaves differently on the two machines.

Vary Test Configurations

There are hundreds if not thousands of configuration variables you could consider:

•Javascript or cookies enabled/disabled in a browser.

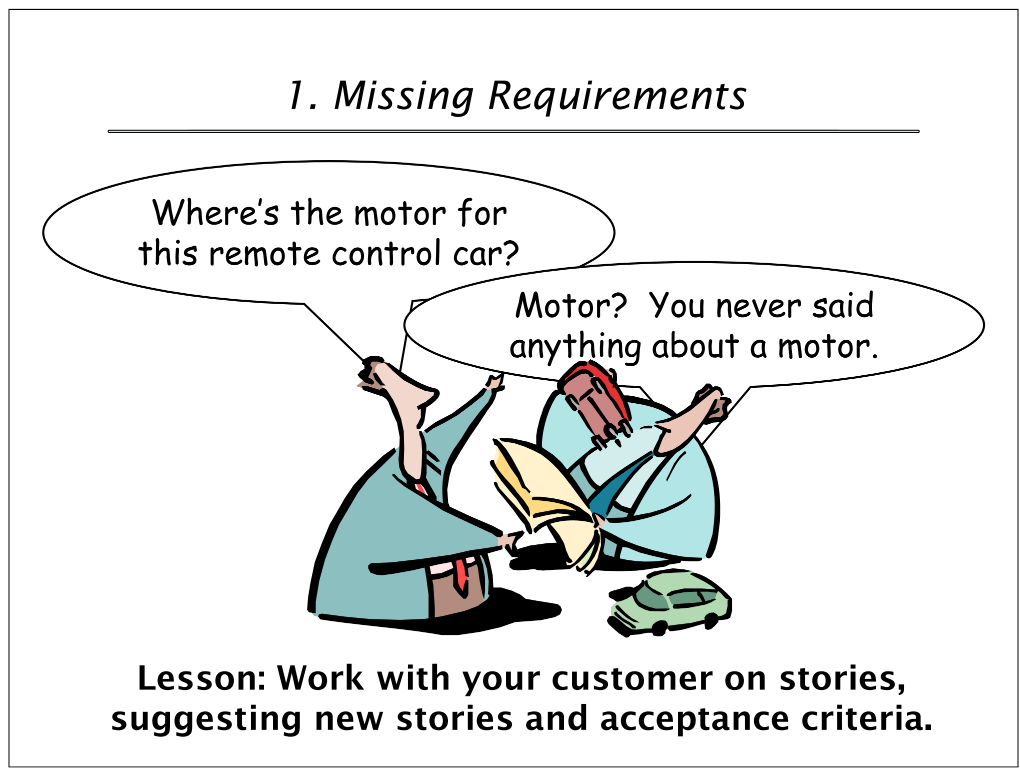
•Security levels and absence of Admin rights.

•Hardware configurations such as mouse configured for left v. right, with and without a mouse wheel, multiple pointing devices or keyboards, presence/absence of a default printer, single/multiple monitors, video driver.

•Network configurations such as presence/absence of a firewall, presence/absence of wireless connections, multiple MAC or IP addresses.

•Software and operating system configurations such as location of programs, color scheme, toolbar placement, language/region, timezones.

 Copyright (c) 2007 Quality Tree Software, Inc.



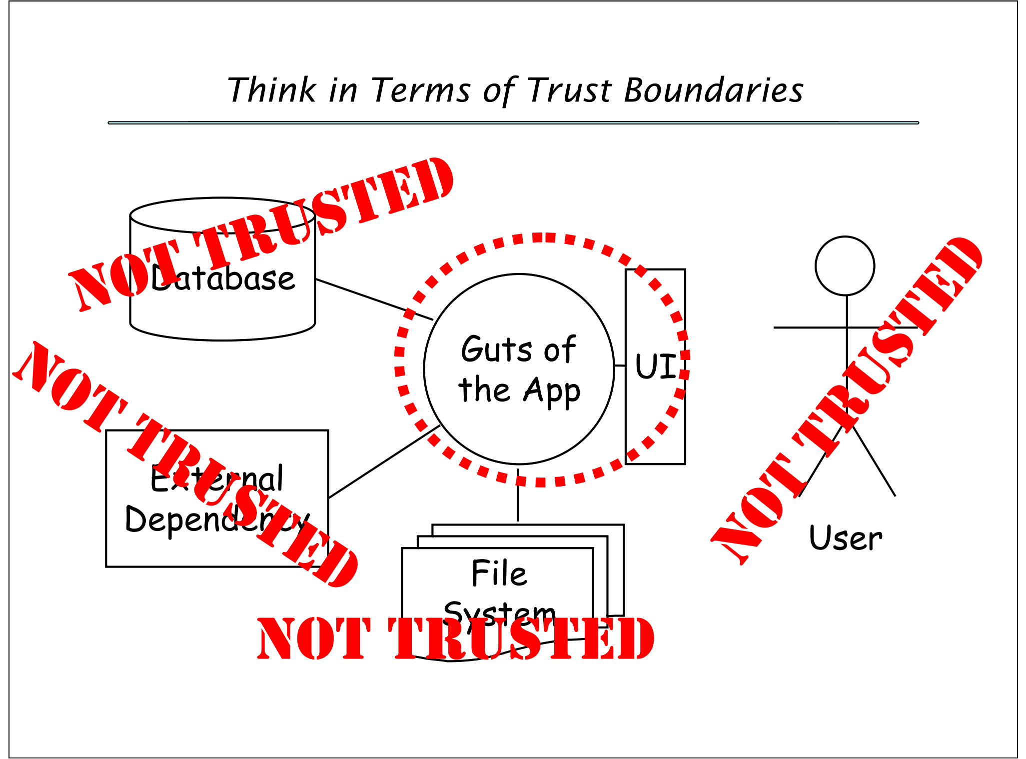
v 1.0. Copyright (c) 2007 Quality Tree Software, Inc.

v 1.0. Copyright (c) 2007 Quality Tree Software, Inc.

Stronger Assertions

What does this really tell us about the object under test?

```
assert !foo.nil?
```



Testing Across Trust Boundaries

Rebecca Wirfs-Brock explains the concept of Trust Boundaries in her book *Object Design: Roles, Responsibilities, and Collaborations*. The concept is simple. In object oriented design, you have collaborators: objects/classes that work together to implement some aspect of the system. So you might have a Customer object that collaborates with an Account object. For two objects inside a trust boundary, the developer can say, "I don't need to put all kinds of error handling and condition checking code here because I know that I can trust that other object over there not to do anything stupid. It won't send this object a null pointer, it won't send this object corrupted data, and it won't make invalid requests of this object." So the notion of a Trust Boundary simplifies design and implementation.

But it also gives you a way to think about what deserves extra testing, to the point of paranoia.

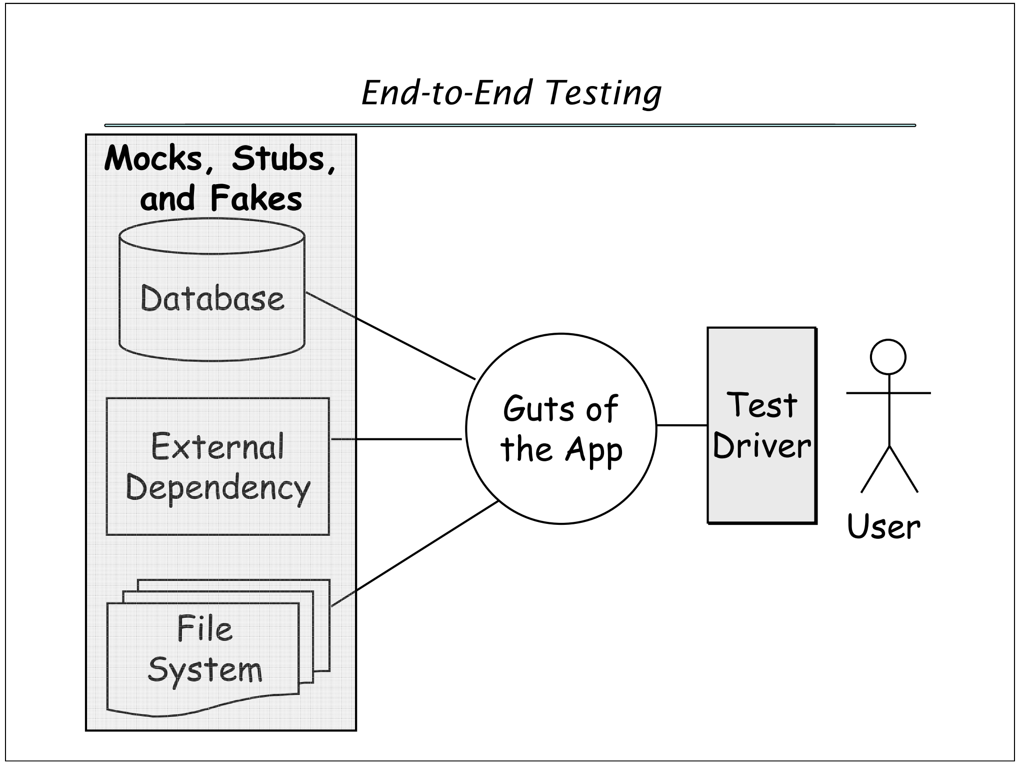
Whenever the software saves data to or retrieves data from a relational database, it's crossing a trust boundary. You never know when someone will have made an index enforce uniqueness or mucked about in the data using raw SQL.

Whenever the software connects to another component over a network, it's crossing a trust boundary. You never know when that network connection will become flaky, nor can you control the bandwidth or latency on the network.

Whenever the software interacts with the file system, it's crossing a trust boundary. You never know when some other process will have a file locked, when a file will have become corrupted, or when an entire directory might have been deleted by a cleanup script run amok.

In short, whenever a component in a system reaches out to an external resource, it's crossed a trust boundary, and it's worth testing to see how it handles the unexpected. And whenever you're testing through an external interface, whether a user interface or an API or a SOAP/XML interface, you're crossing a trust boundary. Systems simply cannot afford to trust either the competence or the good intentions of whatever is on the other side of that external interface.

v 1.0. Copyright (c) 2007 Quality Tree Software, Inc.



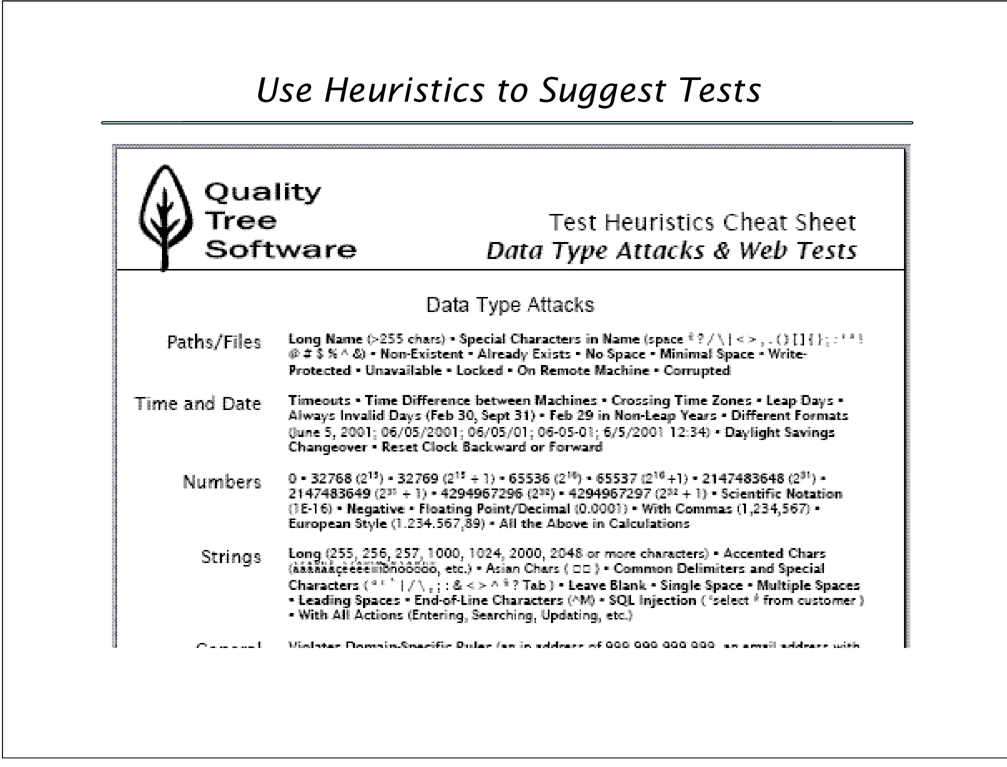
There is No Substitute for End-to-End Testing

End-to-end automated tests tend to be difficult to write, difficult to maintain, cumbersome, and fragile. Writing tests against something as fluid and difficult to test as a user interface is often a recipe for pain. And end-to-end manual testing is time-consuming and often inconsistent.

But the alternatives are worse.

End-to-end automated and manual tests give us important feedback about what the user will actually experience. No matter how green the code-level unit and integration tests are, the testing isn't done until you see it work all the way through to the external interfaces and back again.

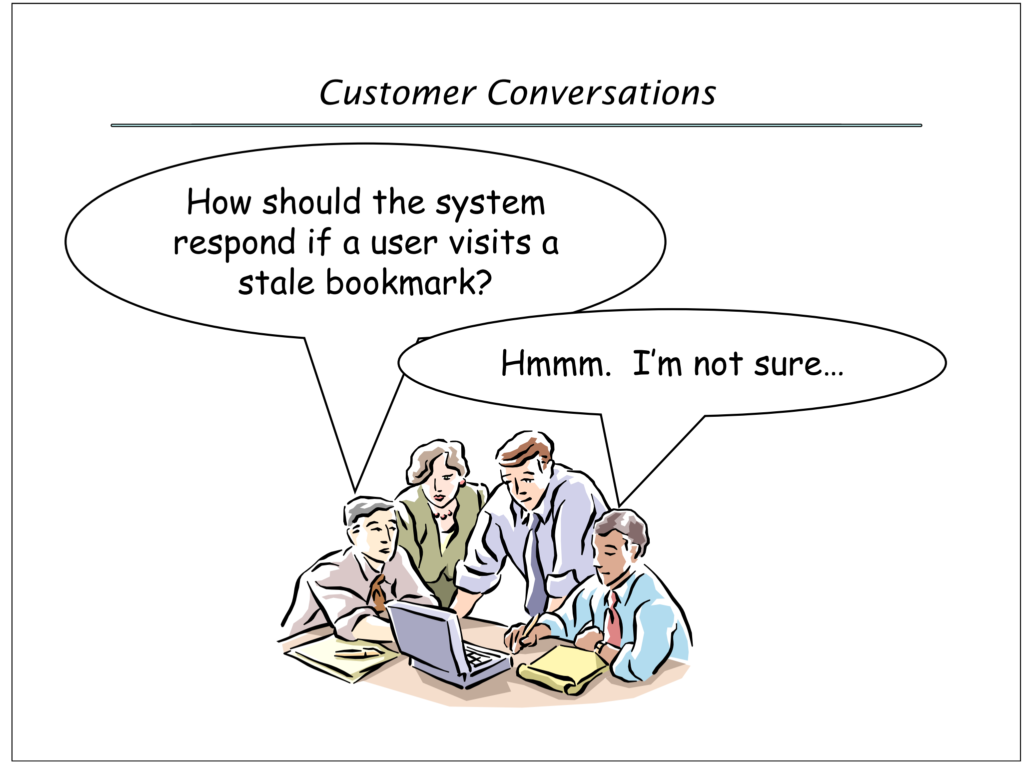
 Copyright (c) 2007 Quality Tree Software, Inc.



Other Test Catalogs

Brian Marick's Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing has a testing catalog in the appendices.

James Whittaker's *How to Break Software* and *How to Break Software Security* contain a laundry list of attacks. (Though Whittaker's work tends to be Windows-centric.)



Use Test Ideas to Prompt Customer Conversations

When thinking about tests, you'll identify potentially interesting conditions, actions, sequences, configurations, and such. Professional testers are particularly good at this. Professional testers know how to take a hand-wavy set of statements like "users belong to groups; groups have permissions; permissions allow for create, read, update, and/or deleting of floozibitzes" and turn them into 193 test cases.

The same skills that enable a tester to come up with 193 test cases out of a one sentence story can help prompt important conversations with your customer that will plug gaps in the requirements. It's just a matter of turning around the heuristics we use for defining tests and using them to ask the customer questions to help them think through their real requirements.

"I see that we have stories to cover creating, reading, and updating floozibitzes," you might say. "But what about deleting?"

Or, "I could imagine a user having bookmarked a detail page for a floozibitz. What if they click that bookmark after the floozibitz has been deleted? What should the system do?"

Or, "Who should have permission to delete a floozibitz?"

The bottom line: any escaped bug on an XP project can be traced back to an inadequate test or incomplete requirements. Tests and stories are two sides of the same coin. And the skills that make us better testers also make us better at requirements elicitation, and thus more likely to ask the right questions, and write the right tests, to reduce the risks.

Thanks!!

Questions? Comments?

Elisabeth Hendrickson
Quality Tree Software, Inc.
www.testobsessed.com
www.qualitytree.com
esh@qualitytree.com

v 1.0. Copyright (c) 2007 Quality Tree Software, Inc.