Exploratory Testing in an Agile Context

A guide to using Exploratory Testing on Agile software development teams.

Elisabeth Hendrickson

What is Exploratory Testing Anyway?

Exploratory Testing. So you bang on the keyboard randomly, right?
Um no, not really.

Exploratory Testing involves ***simultaneously learning about the software under test while designing and executing tests, using feedback from the last test to inform the next.***

Cem Kaner coined the term in the 1980's, though the practice undoubtedly started much earlier.

Because there is a common misunderstanding that Exploratory Testing means simply "do random stuff and see what happens," it's important to emphasize that this is a rigorous investigative practice. We use the same kinds of test design analysis techniques and heuristics in Exploratory Testing that we do in traditional test design, but we execute the tests immediately. The test design and execution becomes inseparable, a single activity.

Tests as Experiments

When we design tests while exploring, we have a hypothesis about how the software will behave: perhaps we suspect the software will exhibit a particular type of error, or perhaps we are seeking to confirm how the software works.

Either way, we think of a little experiment, and then perform it immediately. Our experiments teach us about the system. We learn how it works, what makes it tick, how it’s organized. We discover not just what works and what doesn’t, but also general patterns of vulnerabilities.

The more we learn about how the system works, and the circumstances under which it misbehaves, the better we are at designing good experiments that yield useful information.

Really? No Keyboard Banging?

Sometimes we do things when exploring that seem odd to an outside observer.

James Bach talks about his “Shoe Test” in which he places a shoe on the keyboard.

But James doesn’t put a shoe on the keyboard because he’s trying to come up with wacky random stuff. He does it because he has noticed that some software exhibits bad behavior when it receives too many key inputs at one time. Placing a cat on the keyboard, or handing the keyboard to a 2-year-old, might result in similar behavior. But a shoe or a book is usually more handy.

So yes, there might be keyboard banging involved in Exploratory Testing. But it’s keyboard banging with a theory of potential error, not just random wacky stuff because we can't think of anything better to do.

Exploratory Testing and Agile

Exploratory Testing is a core testing practice for Agile teams.

In an Agile context, scripted regression tests are typically automated. The Continuous Integration (CI) system executes these tests with every build. Such automation is essential to obtaining the fast feedback that Agile teams need to deliver frequently at a sustainable pace.

However, all that automation is not sufficient to ensure that the system does everything we expect and nothing we don't. Someone, somewhere, has to actually use the system.

And yet writing detailed test scripts for a manual test effort doesn't make sense in an Agile context. If it's a test that's important enough to script, and execute repeatably, it's important enough to automate. [1]

So Agile teams still need manual testing, but don't need repeatable-follow-a-script testing. Instead, Agile teams need a manual testing approach that's adaptable (because the software under test changes

[1] Discussing automation strategies is outside the scope of this book. But yes, I do think that if you can write a manual script for a test, you can automate it. If you can give a human repeatable step-by-step instructions for executing a test, you can write an automated test to do the same thing. And if you're going to argue "but you can't write software to unplug a network cable" then I'm going to tell you that you can. Or more specifically, the programmers who write the network error handling logic can make it possible by writing the code such that a simulator can fake pulling the network cable. You'll still need to do manual exploration, of course, but having a small number of repeatable automated regression tests around network cable unplugging will save the team a world of grief on the day that someone accidentally checks in code that breaks that error handling.

very quickly on Agile projects) and that produces large amounts of information quickly (because Agile teams thrive on fast feedback).

Exploratory Testing is a perfect fit.

Instead of documenting step-by-step instructions, we capture just simple charter statements that represent questions we want testing to be able to answer.

Then using charters to focus our explorations and heuristics to guide us, we move through the software rapidly, poking and prodding to reveal unintended consequences of design decisions and risks we didn't consider in advance. In doing so, we gather large amounts of information very quickly.

Further, because Exploratory Testing involves using the results from the last test to inform the next, we can adapt our testing approach quickly if we notice something that might indicate a vulnerability or risk. We can tailor our investigations to what's important right now, without being constrained to follow a test script that we thought was important some number of months ago.

This book covers the essential elements of Exploratory Testing: **learning** the system from the outside in; **designing** tests using **heuristics**; **executing** tests and **observing** results closely; and **integrating** Exploratory Testing into story development within a Sprint or Iteration.

Learning.

In which we discuss learning about the system and capturing our understanding for later use in designing tests.

Simple Scenarios

What is the software intended to do?

This is where you start learning the system: with the simplest case that should work. You’re not testing the software yet, but you are testing your understanding.

Are you exploring a shopping cart? Add an item. Check out. Testing a word processor? Write some text. Add an image. Save it. Testing a DNS server? Query it to get an IP address for a domain. Whatever it is that you’re testing, start by doing basic user actions to see what the system does.

As you go, start taking notes about...

Nouns and Verbs

The nouns of the system are the things that it manipulates, manages, and interacts with. Typical nouns for a shopping cart would include things like: carts, registered users, items, quantities, prices, discounts, coupons, payment methods, and addresses.

The verbs of the system are the actions you can take using the system, or that the system can take. In the shopping cart example verbs could include: checkout, update, remove, abandon, save, login, logout.

As you collect a list of nouns and verbs for the system, you’re capturing a language that describes the system. This will come in handy for designing tests, as well as for further explorations to learn more about the shape of the system.

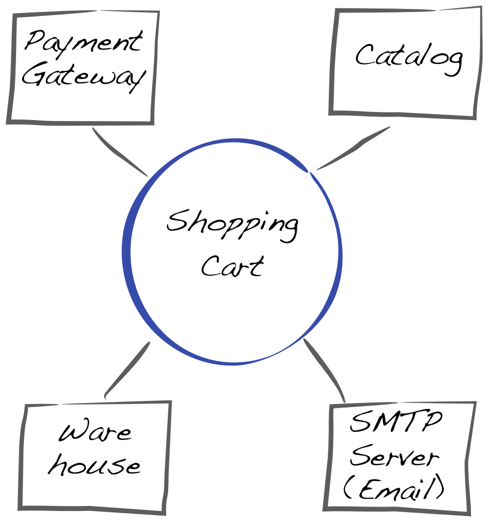
Alternate Paths

As you explore the system noting nouns and verbs, you're likely to notice that there is more than one way to do things.

Perhaps you can login by bringing up a Login page, and also by filling in your login information into username/password fields that appear at the top of each page. Perhaps there's a "remember me" option that allows the system to keep you logged in for days or weeks.

Perhaps you can delete an item from the shopping cart by changing the quantity to 0, or by clicking a Delete button, or by emptying your shopping cart.

Whenever you notice that there are multiple ways to get to the same result, make a note. You'll want to vary these paths as you explore.



Context Diagrams

Software lives within an entire ecosystem. It runs on an operating system, stores data on the file system or in a database, and integrates with other software, like payment gateways for credit card processing, SMTP servers for email, and internal systems like sales reporting, inventory control, and warehouse/shipping applications.

As you begin exploring, create a simple context diagram that shows external resources or dependencies, like the one to the left. Over time, when you learn more about the system and the context within which it lives, add more details.

Every dependent system provides additional opportunities for exploration.

Variables

If you're a programmer, a variable is a named location in memory. You declare variables with statements like "int foo;"

That's not the kind of variable we're talking about here.

Rather, we're talking about "variable" in the sense of "things you can vary." More specifically, a variable in testing is anything that you can change, or cause to be changed, via external interfaces (like the UI or the file system), that might affect the behavior of the system.

Sometimes variables are obviously changeable things like the value in a field on a form. Sometimes they're obvious, but not intended to be changed directly, like the key/value pairs in a URL string for a web-based application. Sometimes they're subtle things that can only be controlled indirectly, like the number of users logged in at any given time or the number of results returned by a search.

Subtle variables are the ones we often miss when analyzing the software to design tests. Consider for example the Therac-25 case that Nancy Leveson describes in her book *Safeware*.

The Therac-25 was a radiation therapy machine that overdosed patients with radiation, killing them. The story dates back to the 1980's, but it's still relevant today.

According to Nancy Leveson, in at least one case, the malfunction that caused a death could be traced back to the medical technician's entering and then editing the treatment data in under 8 seconds. That's the time it took the magnetic locks to engage. Notice two key subtle variables, or things that a user could change: the **speed of input** and the **sequence of actions**.

Further, Leveson found that every 256th time the setup routine ran, it bypassed an important safety check. That's yet another subtle variable: **the number of times the setup routine ran.**

As you begin to learn the system, make note of things you notice that you can change, or cause to be changed, that are subtle or otherwise interesting.

The more you look for variables, the more you'll find. They're everywhere.

Then, as you explore, use the Heuristics described in the Test Design section to suggest interesting variations.

Designing.

In which we discuss using heuristics to design tests on the fly.

Using Heuristics

You've started exploring the system. You have the general shape of it. You understand the context in which it lives, the kind of data it manipulates, the actions you can take. You may have some ideas about sequences and configurations that might be interesting.

This section lays out a selection of heuristics: test ideas that are general enough to be applicable across a wide range of domains and technologies.

Use this section as a reference when you run out of new ideas for things to try.

[note: each heuristic will get about a page worth of explanation/ examples.]

File System Interactions

Network Variations

CRUD: Create, Read, Update, Delete

Position: Beginning, Middle, End

Count: 0, 1, Many

Interruptions

States and Transitions

Time: Before, During, After

Combining Heuristics

Randomizing Noun and Verb Combinations

Following the Data

Personas

Executing.

In which we discuss observing, note-taking, and defect isolation.

Mechanics.

In which we discuss how to structure your explorations within a Sprint or Iteration in an Agile process.

Charters

Traditional scripted testing is documentation-centric with written Test Plans, Test Strategies, Test Cases, and Test Procedures.

Exploratory Testing involves far less documentation. We don't document each and every test case. Instead, we write charters: simple statements of the information that we hope to discover through exploration.

One way of expressing charters is with the simple template:

> **Explore *area***
> **With *resources, constraints, tools, etc.***
> **To discover *information***

Some charters are broad: "Explore the system with typical usage scenarios to discover how it works."

Some are narrow: “Explore the File Import feature with various invalid file formats to discover if there are any circumstances under which the error handling does not give a reasonable response to an invalid file.”

Some are about valid usage scenarios with representative user personas: “Explore browsing and shopping with a non-technical user perspective to discover how easy or hard it is to buy items.”

Others are about misusage scenarios: “Explore the shopping cart feature with a tool to perform http POST requests to discover if there’s any way to get stuff for free.”

We might express charters a little differently, and that’s fine. The purpose of the charter is to provide a focus to our explorations.

Sessions

During a session, if we notice that we're going off on a tangent, we use the charter to remind us what we're supposed to be investigating. We might make a note of another charter to investigate later, but we make a point of staying on charter for now.

Frequently Asked Questions.

In which we tackle questions that seem to come up over and over (and over) again.

References

Bach, James. “What Is Exploratory Testing?”

Bach, Jonathan. “Session-Based Test Management”

Kohl, Jonathan. “Exploratory Testing on Agile Teams”

Kohl, Jonathan, “User Profiles and Exploratory Testing”

Marick, Brian. “A Survey of Exploratory Testing”

Tinkham, Andy and Kaner, Cem. “Exploring Exploratory Testing”

If you liked this book...

You'll love our classes...

[blatant promotional text goes here.]