

Exploratory Testing in an Agile Context

Elisabeth Hendrickson

Quality Tree Software, Inc.

www.qualitytree.com

esh@qualitytree.com

To contact me after this session,
text WSA to INTRO (46876)

Chapter 1: Explorations

“Exploratory Testing” is a style of testing in which we learn about the behavior of a system by designing a small test, executing it immediately, using the information gleaned from the last test to inform the next. We continue that rapid cycle of design a test, execute it, observe until we've characterized the capabilities and limitations of the system with respect to a charter, or mission.

Let's examine each of these three key aspects in more detail.

Exploratory Testing is:
Simultaneously learning about the system while designing and executing tests, using feedback from the last test to inform the next.

1. Designing.

Good testers know a lot about test design. Test design involves identifying interesting things to vary, and interesting ways in which to vary them. We can design tests around data: boundaries, special characters, long inputs, nulls, number fields with 0, etc. We can also design tests around sequences and logic using state models, sequence diagrams, flow charts, decision tables, and other design artifacts. All the traditional test design techniques apply. The difference is that while we might make notes about the tests we're designing, we aren't writing down formal test cases.

2. Executing.

As soon as we think of a test, we execute it. This is what distinguishes exploratory testing from other styles of testing. We're not squirreling away a large set of designed tests for some future

time when we (or someone else) will execute them. We execute them immediately.

3. Observing / Learning.

We're observing what the system does and does not do. We discover how it operates. We find its quirks and peculiarities while characterizing its capabilities and limitations. Further, we're learning the system well enough that we can reflect our observations back to the rest of the team. The information that our explorations uncover will help to move the project forward, but only if we can report it out in a way that the rest of the team can digest.


$$\begin{array}{r} \text{Checked} \\ + \text{Explored} \\ \hline = \text{Tested} \end{array}$$

Two Sides of Testing: Checking and Exploring

There is a perpetual ongoing debate in the testing community: Does "good testing" emphasize executing a comprehensive set of scripted tests designed from the requirements of the system? Or should we

instead take an exploratory approach to discovering the real risks in the system?

The answers to these questions have historically divided the testing community and the debate has sometimes become downright rancorous with each side accusing the other of irresponsible practices that increase risk and decrease quality.

However, the debate represents a false dichotomy. Agile teams do not do one or the other; they do both.

Agile practices like TDD and ATDD result in automated regression code-facing and business-facing checks. We run these checks as part of the CI build so that they tell us any time a change results in a previously met expectation being violated.

For example, if we are working on a story around editing profiles, we might have an acceptance test like this Cucumber scenario:

Given I am not logged in
When I visit the Edit Profile page
Then I should be redirected to the Login page
When I log in
Then I should be redirected to the Edit Profile page

If we ever made a change that inadvertently caused any aspect of this behavior to change, we would want to know. Even if we hadn't automated this test, it is a check that we would want to run all the time.

However, there can be any number of unintended consequences that acceptance tests would not expose. This acceptance test will ensure that a user has to be logged in to edit their profile. But what if users who are logged in can edit any other user's profile as well? These are the kinds of risks and vulnerabilities that exploratory testing can reveal.

Because an agile development can accept new and unanticipated functionality so fast, it is impossible to reason out the consequences of every decision ahead of time.

In other words, agile programs are more subject to unintended consequences of choices simply because choices happen so much faster. This is where exploratory testing saves the day. Because the program always runs, it is always ready to be explored.

...

[T]here is a tremendous opportunity in front of us if we are just willing to support exploratory testing with the same vigor that we now support automatic testing.

--Ward Cunningham

*Posted March 25, 2004 by on the agile-testing mail list
(see <http://groups.yahoo.com/group/agile-testing/message/3881>)*

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.

Note that each of our experiments involves a hypothesis: a prediction about the resulting behavior of the system. We might start with a hypothesis that the system will be able to handle what we are throwing at it. Or we might start with a theory of error: a guess about the risks we're about to uncover. Either way, each experiment is deliberately designed to conform our refute our hypothesis.

The idea that exploratory testing is a deliberate and thoughtful process surprises some.

Really? No Keyboard Banging?

Sometimes we do things when exploring that seem odd to an outside observer. For example James Bach talks about his “Shoe Test” in which he places a shoe on the keyboard.

*Testing
Tool?*



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.

Unbreakable

I first visited Atomic Object in Grand Rapids, Michigan a few years ago. After initial introductions, Karlin Fox took me over to a workstation and introduced me to a system that they had recently put into production.

The system was a kiosk to be installed in home improvement stores that would allow users to see a preview of a new paint color in rooms in their homes. Users could upload a picture of a room from external media such as a CD or USB drive, then manipulate the picture, changing the color of the walls.

Karlin suggested that I explore.

I started by acting as a consumer. I experimented with uploading various pictures and painting the walls. I quickly realized that I would not find anything surprising on the happy paths.

Next I channelled my inner 3 year old and tried interrupting the system by pulling out the media and hitting keys randomly. For every scenario I tried, the system responded in a reasonable way.

Undaunted, I tried corrupting data, providing invalid file formats, and potentially risky file names.

I have explored numerous systems produced by high caliber teams over the years. Usually I find any number of bugs, often serious ones. Much to my surprise and delight, I could not find anything that I considered a defect in this kiosk application.

Curious as to how the team had created software that I could not break, I asked about the process the team used. Karlin told me that they had an exploratory tester on the team.

"He drove us nuts!" he reported. "Always finding stuff." Karlin paused. Then his face lit up. "It was great!"

Chapter 2: Organized Explorations: Charters and Sessions

In a letter dated June 20, 1803, Thomas Jefferson, the 3rd President of the United States, gave the explorers Lewis and Clark a mission:

The Object of your mission is to explore the Missouri river & such principal stream of it as by its course and communication with the waters of the Pacific ocean, whether the Columbia, Oregon, Colorado or any other river may offer the most direct & practicable water communication across this continent for the purpose of commerce.

The content of the objective that Jefferson set forth is significant. Notice that Jefferson specified the areas he wanted explored: the Missouri river and connecting waterways to the Pacific. Notice also that he specified "for the purpose of commerce." He was interested in a passageway for commerce; he was not particularly interested in nice picnic spots or locations for future national parks.

Earlier in that same letter letter, Jefferson told Lewis and Clark what they would have to work with:

Instruments for ascertaining, by celestial observations, the geography of the country through which you will pass, have already been provided. Light articles for barter and presents among the Indians, arms for your attendants, say from ten to twelve men, boats, tents, and other traveling apparatus, with ammunition, medicine, surgical instruments and provisions, you will have prepared, with such aids as the secretary at war can

yield in his department; and from him also you will receive authority to engage among our troops, by voluntary agreement, the attendants above mentioned; over whom you, as their commanding officer, are invested with all the powers the laws give in such a case.

Exploring software has much in common with exploring territory:

- There are any number of surprises and adventures awaiting us (including bugs).
- We can use tools to help us; however, the most important tool we carry is the one between our ears.
- Sometimes exploring is a fun romp, sometimes it's a slow slog, and sometimes we tread on treacherous ground.
- Finally, if the map and the territory differ, believe the territory.

Thus it should come as no surprise that the contents of the charter that Jefferson gave Lewis and Clark can serve as a model for the charters we'll create for exploratory testing.

Jefferson specified where to explore, the resources the explorers would have to work with, and the information that they were to seek:

*Explore area
with resources
to discover information*

In the case of Lewis and Clark, they were to:

Explore the Missouri river & such
principal stream...

With instruments for ascertaining, by
celestial observations, the geography
of the country...

To discover direct & practicable water
communication across this continent...

In testing software, we have different objectives. We will be exploring some externally visible behavior such as a feature in a system. We might explore using tools or resources such as a specific data set. However, like Lewis and Clark, our ultimate goal in exploring is to discover information that will help move the project forward.

Our charters look like:

*Explore a story, feature, area, or system
with resources, constraints, heuristics,
or other features
to discover information*

For example, imagine we have a story like:

*As a user, I want to update my personal
information on my profile so that my
public profile stays up to date.*

And we might have charters like:

Explore editing profiles
with sql and javascript injection attacks
to discover security vulnerabilities

Explore editing profiles
with the authentication feature
to discover surprises

Explore editing profiles
with different kinds of users
to discover interactions between profile
editing and roles

Good Charters

A good charter offers direction without over-specifying test actions. As an example, the following isn't a charter; it's a test case.

Too Specific

*Explore editing last name
with the value "O'Malley"
to discover if the profile
edit feature can handle
names with apostrophes*

When charters are too specific, they become just a different way of expressing individual tests. If they are tests that we need to repeat with every iteration, they should be automated.

By contrast, charters that are too broad run the risk of not providing the most important information. Like a story that is too big and ambiguous, we don't know how to tell when we're done.

For example, the following charter calls for exploring the entire system with a large and undefined set of resources. The result is so vague that we could spend weeks investigating it and still not be sure we discovered all the important risks and vulnerabilities.

Too General

*Explore system security
with all the hacking
programs we can find
to discover any security
holes*

It is better to focus on a single area and/or specific types of security holes with each charter. A better set of focused security-related charters might be:

*Explore input fields
with javascript and sql injection attacks
to discover security vulnerabilities*

*Explore pages requiring login
with spoofed URLs and POST parameters
to discover if users can gain access to
content they did not pay for*

About The Charter Template

The formulaic "Explore/With/To Discover" is like the user story format of "As A / I want / Because." It reminds us of the information that we need.

However, sometimes the phrasing is awkward. It can feel constraining rather than supporting if the ideas we want to express do not fit neatly into the structure. We want to say:

"Play with buffer overflow exploits to try to gain shell access to the server"

If we constrain ourselves to fit every charter into the "explore/with/to discover" format we end up with the much more awkward:

*Explore inputs
with buffer overflow exploits*

*to discover if there is any way a user could
gain shell access to the server*

As with any template, this format for phrasing charters is just a guide. It appears throughout this document, but please do not interpret that as a “one right way” to express charters. The important thing about charters is that they focus our efforts without constraining our thinking.

Chapter 3: "Recon"

In their "ET with Subtitles" video, James and Jon Bach perform what they call a "Recon" session on an "Easy Button," a novelty toy with a single button that, when pressed, prompts the device to say, "That was easy."

This recon session is a special kind of session in which we begin mapping the territory of the system.

In the first session our charter is:

*Explore Scrabble Flash®
with everything in the box
to discover how it works*

Since we just discussed charters that are too general in the last section, it may seem that this charter contradicts the advice on focusing charters on a given area, resource, or type of information.

However, recon is when we begin learning about:

The ecosystem in which the software under test resides. What does the software we're testing connect to? What system resources does it use?

Touchpoints. Identifying public or private interface that provides visibility or control. These touchpoints provide places to provoke, monitor, and verify the system.

Variables. The things we can change, or cause to be changed. Some variables are obvious, like system inputs and outputs, or configuration settings. Others are much more subtle such as sequences of actions, timing, attributes of data in the system.

Obvious vulnerabilities and potential risks. These will begin to suggest charters for future sessions.

The recon session sets the stage for all the other sessions and missions.

Chapter 4: “Variables”

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

```
int foo;
```

In this case however, we're talking about the garden-variety English word “variable,” or something that can change.

More specifically, a variable in testing is anything that we can change, or cause to be changed, via external interfaces (like the UI, database, or 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, at least one 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 that there is a subtle but crucial variable at play here: speed of input. If the technician was fast enough, they could enter the treatment before the safety locks engaged.

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.

Our second charter in this workshop is:

*Explore Scrabble Flash®
With everything in the box
To discover variables*

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

As we discover variables, we can use test design heuristics to figure out how to manipulate them. Then, as we explore, use the Heuristics described in the Test Design section to suggest interesting variations.

If we discover variables involving counts of objects, we can use the "0, 1, Many" heuristic. If we discover variables involving selection, we can use "Some, None, All." If we discover that timing is a variable, we can use the "interrupt" heuristic.

(For a list of these heuristics, please see the "Test Heuristics Cheatsheet" in the appendix at the end of this document.)

Discovering variables leads to the follow on charter:

*Explore Scrabble Flash
With the variables you discovered
and test heuristics
To discover surprises*

Chapter 5: Third Session - "Timing and Sequences"

"All models are wrong. Some are useful." --George Box

In this third and final session of this workshop, we will exercise just one class of variables that you (no doubt) discovered in the second session: timing and sequence related variables.

The Scrabble Flash game is particularly well suited for state analysis because so much of its functionality involves timing. The games are timed. The tiles communicate with one another, and communication always involves timing.

Exploring States

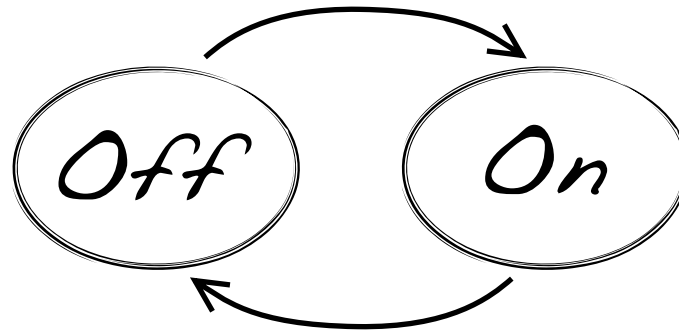
State modeling offers an analysis tool for discovering surprises related to timing.

In exploring, we map the states we observe. We have no way of knowing whether the states we identify are expressed as a state model in the code. And we have no way of knowing if our list of states is complete. It doesn't matter.

The model we are building is a tool to prompt test ideas.

The World's Simplest State Model

Here we see the world's simplest state model. It has two states: On and Off. And indeed this model could describe absolutely any software or electronic device.



At some point the thing is off. And then at some point it comes on. And then it goes off again. Every single electronic system from a FaceBook widget to an electronic book reader to a word processor to a Scrabble Flash tile could be described by this simple model.

So what good is it? What is the point of thinking about such a trivial generic model?

Consider all the ways that a Scrabble Flash tile can go from on to off, and from off to on. It's quite a list. A tile turns on by pressing the button on the front. It can turn off by timing out, by pressing

the button on the front again, by pressing its neighbor's button, by running out of battery power, or by resetting it.

Now imagine that we are in the middle of a game. Letters are displaying on the tiles in a 5-tile, 2-person game. Consider all the interesting ways to turn a single tile off. Now consider: what happens to the rest of the game? There are any number of interesting experiments to run with turning tiles off mid-game.

All those ideas can stem from a single simple model.

Identifying States

One of the biggest challenges with mapping states when we're exploring is knowing when we've found one worth putting on our map.

It helps to think of states as situations in which the system exhibits a distinguishable set of behaviors: a behavioral mode.

If we find a circumstance where it makes sense to say “While the game is _____” or “While this tile is _____” we’ve found a state. For example: “A tile shows spinning blocks while it is waiting to connect to other tiles.”

For example, when Scrabble Flash is in the middle of a game, the tiles show letters. We can say “while the tiles display letters...”

We can prompt a state transition by arranging the tiles in the correct sequence. At that point the tiles flashing in acknowledgement, then display a new set of letters (as long as there is time remaining in the game).

Note that this is very different behavior from when Scrabble Flash is waiting for the player to choose a game. Then, changing the sequence of the tiles has a different result.

Thus, we can recognize states by asking the question:

- What can I do now that I couldn't do before?
- What can't I do now that I could do before?
- Do my actions have different results now?

Events Trigger State Changes

Some states are transitory, like a start up sequence. The trigger to move from one state to another depends entirely on the inner workings of the tiles. It might be time (as with a timeout) or it might be some internal process completing.

Other states are triggered by user actions (rearranging the tiles or pushing the buttons). We can control these state changes directly.

Both transitory and user-controlled states, and the events that prompt the transitions to and from those states, are interesting to explore.

To identify events, consider what triggers state transitions, such as:

- User actions: clicking links or pushing buttons
- Changing conditions: a processing task completed
- Application interaction: another application issues start and stop requests
- Operating system requests: shutdown
- Time: a session timeout.

Exploring Scrabble Flash with States

In our next session, we'll map states and transitions in Scrabble Flash:

*Explore Scrabble Flash
With states and transitions
To discover surprises*

There once was an Indian medicine man whose responsibilities included creating hunting maps for his tribe. Whenever game got sparse, he'd lay a piece of fresh leather out in the sun to dry. Then he'd fold and twist it in his hands, say a few prayers over it, and smooth it out. The rawhide was now crisscrossed with lines and wrinkles. The medicine man marked some basic reference points on the rawhide, and--presto!--a new game map was created. The wrinkles represented new trails the hunters should follow. When the hunters followed the map's newly defined trails, they invariably discovered abundant game.

Moral: By allowing the rawhide's random folds to represent hunting trails, he pointed the hunters to places they previously had not looked.

--Roger von Oech, A Whack on the Side of the Head

Chapter 6: How Much is Enough?

Sometimes it seems that the more we explore, the more we find to explore. This can become a problem if we don't consider a story done until it has been implemented, checked, and explored. Thus the question naturally arises: how do we know when we're done exploring?

In order to declare anything "done" we need a stopping heuristic. Some possibilities might be that we know we have explored sufficiently when:

We have no more charters left undone for a given story.

Or perhaps:

We find no new interesting information within a specified period of time.

Or even:

Additional information would not help move the project forward.

The decision of what stopping heuristic to adopt rests with the team; it is not a decision that we can make unilaterally.

Appendix: Resources

Agans, D. (2002). *Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*: AMACOM.

Bach, J., & Bach, J. (2009). "ET with Subtitles" video: <http://www.youtube.com/watch?v=Vy0I2SB5OLo>

Beizer, B. (1990). *Software Testing Techniques (2nd ed.)*: Thomson Computer Press.

Gause, D. C., & Weinberg, G. M. *Are Your Lights On? : How to Figure Out What the Problem Really Is*.

Hoffman, D. (1998). "A Taxonomy of Test Oracles". Quality Week Conference. Available online: <http://www.softwarequalitymethods.com/Papers/OracleTax.pdf>

Iberle, K. (2002). "But Will It Work for Me?" Proceedings of the Pacific Northwest Quality Conference. Available online: <http://www.kiberle.com/articles.htm>

Kaner, C., Bach, J., & Pettichord, B. Lessons Learned in Software Testing. Kaner, C., Falk, J., & Nguyen, H. (1999). Testing Computer Software (2nd ed.): John Wiley & Sons.

Kruchten, P. (1995). "Architectural Blueprints—the “4+1” View Model of Software Architecture" (Vol. 12). IEEE Software.

Meyers, G. (1979). *The Art of Software Testing*: John Wiley & Sons.

Michalko, M. (1991). *Thinkertoys (A Handbook of Business Creativity)*: Ten Speed Press.

Oech, R. V. (1990). *Creative Whack Pack (Cards ed.)*: United States Games Systems.

Peterson, I. (1996). *Fatal Defect : Chasing Killer Computer Bugs*: Vintage.

Seashore, C. N. (1997). *What Did You Say?: The Art of Giving and Receiving Feedback*: Bingham House Books.

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



Data Type Attacks

Paths/Files	Long Name (>255 chars) ▪ Special Characters in Name (space * ? / \ < > , . () [] { } ; : ' " ! @ # \$ % ^ &) ▪ Non-Existent ▪ Already Exists ▪ No Space ▪ Minimal Space ▪ Write-Protected ▪ Unavailable ▪ Locked ▪ On Remote Machine ▪ Corrupted
Time and Date	Timeouts ▪ Time Difference between Machines ▪ Crossing Time Zones ▪ Leap Days ▪ Always Invalid Days (Feb 30, Sept 31) ▪ Feb 29 in Non-Leap Years ▪ Different Formats (June 5, 2001; 06/05/2001; 06/05/01; 06-05-01; 6/5/2001 12:34) ▪ Daylight Savings Changeover ▪ Reset Clock Backward or Forward
Numbers	0 ▪ 32768 (2^{15}) ▪ 32769 ($2^{15} + 1$) ▪ 65536 (2^{16}) ▪ 65537 ($2^{16} + 1$) ▪ 2147483648 (2^{31}) ▪ 2147483649 ($2^{31} + 1$) ▪ 4294967296 (2^{32}) ▪ 4294967297 ($2^{32} + 1$) ▪ Scientific Notation (1E-16) ▪ Negative ▪ Floating Point/Decimal (0.0001) ▪ With Commas (1,234,567) ▪ European Style (1.234.567,89) ▪ All the Above in Calculations
Strings	Long (255, 256, 257, 1000, 1024, 2000, 2048 or more characters) ▪ Accented Chars (àáâãäåçèéëîíîïðñòóôö, etc.) ▪ Asian Chars (□□) ▪ Common Delimiters and Special Characters (" ' ` / \ , ; : & < > ^ * ? Tab) ▪ Leave Blank ▪ Single Space ▪ Multiple Spaces ▪ Leading Spaces ▪ End-of-Line Characters (^M) ▪ SQL Injection ('select * from customer) ▪ With All Actions (Entering, Searching, Updating, etc.)
General	Violates Domain-Specific Rules (an ip address of 999.999.999.999, an email address with no "@", an age of -1) ▪ Violates Uniqueness Constraint

Web Tests

Navigation	Back (watch for 'Expired' messages and double-posted transactions) ▪ Refresh ▪ Bookmark the URL ▪ Select Bookmark when Logged Out ▪ Hack the URL (change/remove parameters; <i>see also Data Type Attacks</i>) ▪ Multiple Browser Instances Open
Input	<i>See also Data Type Attacks</i> ▪ HTML/JavaScript Injection (allowing the user to enter arbitrary HTML tags and JavaScript commands can lead to security vulnerabilities) ▪ Check Max Length Defined on Text Inputs ▪ > 5000 Chars in TextAreas
Syntax	HTML Syntax Checker (http://validator.w3.org/) CSS Syntax Checker (http://jigsaw.w3.org/css-validator/)
Preferences	Javascript Off ▪ Cookies Off ▪ Security High ▪ Resize Browser Window ▪ Change Font Size

Testing Wisdom

A test is an experiment designed to reveal information or answer a specific question about the software or system. ▪ *Stakeholders have questions; testers have answers.* ▪ Don't confuse speed with progress. ▪ **Take a contrary approach.** ▪ Observation is exploratory. ▪ *The narrower the view, the wider the ignorance.* ▪ Big bugs are often found by coincidence. ▪ *Bugs cluster.* ▪ Vary sequences, configurations, and data to increase the probability that, if there is a problem, testing will find it. ▪ **It's all about the variables.**

This cheat sheet includes ideas from Elisabeth Hendrickson, James Lyndsay, and Dale Emery



Heuristics

- Variable Analysis** Identify anything whose value can change. Variables can be obvious, subtle, or hidden.
- Touch Points** Identify any public or private interface that provides visibility or control. Provides places to provoke, monitor, and verify the system.
- Boundaries** Approaching the Boundary (*almost too big, almost too small*), At the Boundary
- Goldilocks** Too Big, Too Small, Just Right
- CRUD** Create, Read, Update, Delete
- Follow the Data** Perform a sequence of actions involving data, verifying the data integrity at each step.
(*Example: Enter → Search → Report → Export → Import → Update → View*)
- Configurations** Varying the variables related to configuration (*Screen Resolution; Network Speed, Latency, Signal Strength; Memory; Disk Availability; Count heuristic applied to any peripheral such as 0, 1, Many Monitors, Mice, or Printers*)
- Interruptions** Log Off, Shut Down, Reboot, Kill Process, Disconnect, Hibernate, Timeout, Cancel
- Starvation** CPU, Memory, Network, or Disk at maximum capacity
- Position** Beginning, Middle, End (*Edit at the beginning of the line, middle of the line, end of the line*)
- Selection** Some, None, All (*Some permissions, No permissions, All permissions*)
- Count** 0, 1, Many (*0 transactions, 1 transactions, Many simultaneous transactions*)
- Multi-User** Simultaneous create, update, delete from two accounts or same account logged in twice.
- Flood** Multiple simultaneous transactions or requests flooding the queue.
- Dependencies** Identify “has a” relationships (*a Customer has an Invoice; an Invoice has multiple Line Items*). Apply **CRUD**, **Count**, **Position**, and/or **Selection** heuristics (*Customer has 0, 1, many Invoices; Invoice has 0, 1, many Line Items; Delete last Line Item then Read; Update first Line Item; Some, None, All Line Items are taxable; Delete Customer with 0, 1, Many Invoices*)
- Constraints** Violate constraints (*leave required fields blank, enter invalid combinations in dependent fields, enter duplicate IDs or names*). Apply with the **Input Method** heuristic.
- Input Method** Typing, Copy/Paste, Import, Drag/Drop, Various Interfaces (*GUI v. API*)
- Sequences** Vary Order of Operations ▪ Undo/Redo ▪ Reverse ▪ Combine ▪ Invert ▪ Simultaneous
- Sorting** Alpha v. Numeric ▪ Across Multiple Pages
- State Analysis** Identify states and events/transitions, then represent them in a picture or table. Works with the **Sequences** and **Interruption** heuristics.
- Map Making** Identify a “base” or “home” state. Pick a direction and take one step. Return to base. Repeat.
- Users & Scenarios** Use Cases, Soap Operas, Personae, Extreme Personalities

Frameworks

- Judgment** Inconsistencies, Absences, and Extras with respect to Internal, External – Specific, or External – Cultural reference points. (James Lyndsay, Workroom Productions)
- Observations** Input/Output/Linkage (James Lyndsay, Workroom Productions)
- Flow** Input/Processing/Output
- Requirements** Users/Functions/Attributes/Constraints (Gause & Weinberg *Exploring Requirements*)
- Nouns & Verbs** The objects or data in the system and the ways in which the system manipulates it. Also, Adjectives (attributes) such as Visible, Identical, Verbose and Adverbs (action descriptors) such as Quickly, Slowly, Repeatedly, Precisely, Randomly. Good for creating random scenarios.
- Deming’s Cycle** Plan, Do, Check, Act

This cheat sheet includes ideas from Elisabeth Hendrickson, James Lyndsay, and Dale Emery