Insights into Exploratory Testing

2-day workshop notes, IdeaGen workshop 2 November 2017, Nottingham

James Lyndsay, Workroom Productions

Welcome to *Insights into Exploratory Testing*!

Today's workshop is driven by hands-on exercises, group work and conversations. It is designed to stimulate your thoughts, rather than to provide a single right way of working. You will have new ideas – and ideas that are shared tend to change and grow stronger. Please share your thoughts with each other, listen generously, and offer interesting differences.

This is a workbook, not an instruction manual. It has spaces for *you* to write down *your* results, *your* conclusions, and whatever *you* want to learn from those around you. As a rule of thumb, the more you keep a note of your ideas,the more useful the workbook will be when you return to it.

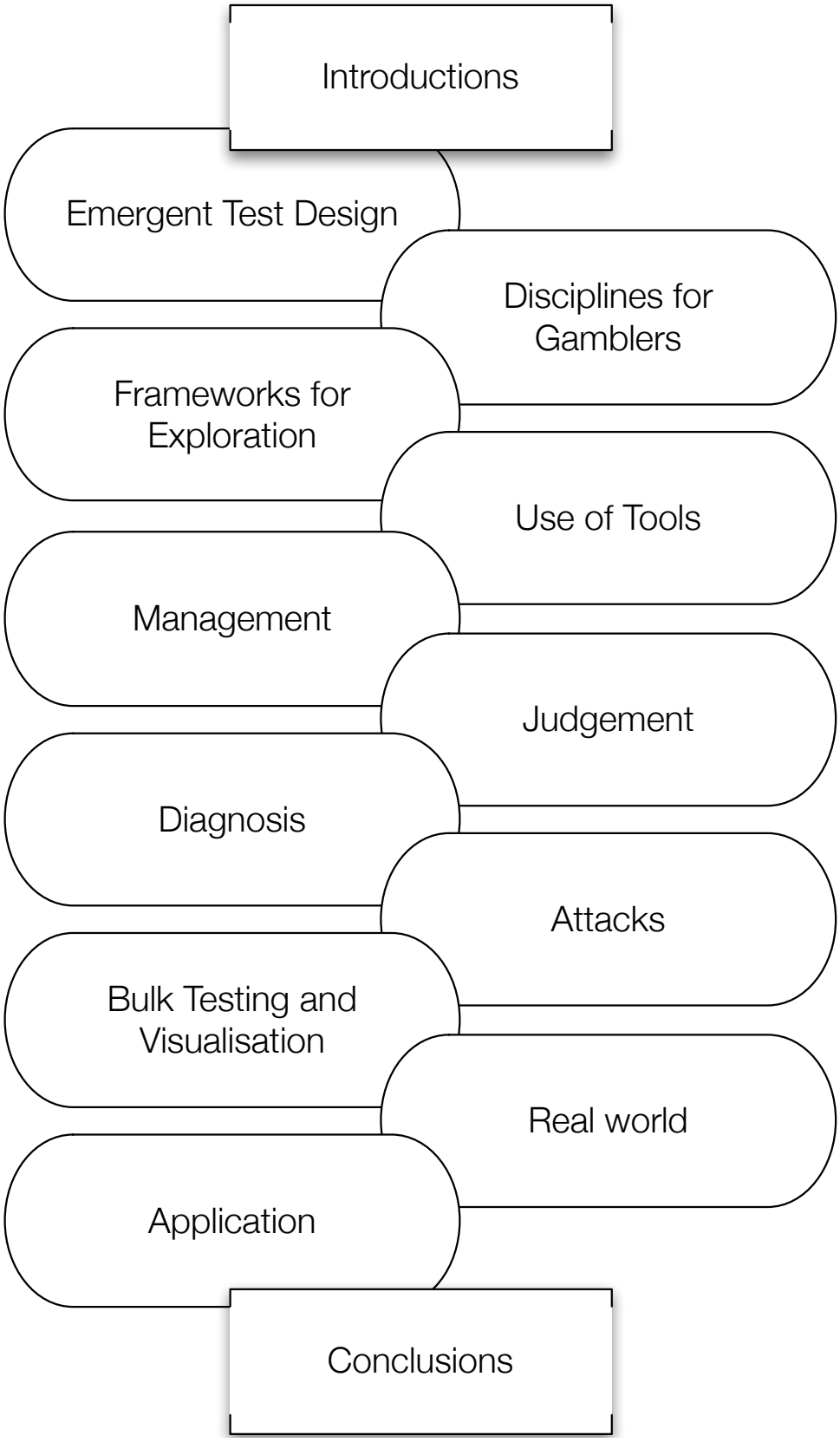
You will need a computer for the exercises. If you're sitting next to someone with – or without – a computer, please share. The exercises are designed for group work, and you should find it illuminating to work together.

You'll find your exercises at `ideagen.workroomprds.com`. You'll need a web browser and the flash plugin to work with the exercises, but the exercises themselves do not install anything on your machine.

Interruptions can disrupt your whole group as you engage in discussions and exercises. Please turn your phone and email application off. We will have regular breaks, and I ask you to do your fellow participants the courtesy of using the breaks for dealing with your urgent and regular work. If someone does have to leave and come back later, please work to re-integrate them into your group.

Map of the workshop

...approximately



... but that's how I'd run it with an empty room.

What we cover, and how we cover it, is up to you (and me, a bit).

We may not cover everything in the map / table of contents.

We will cover plenty of it.

We will spend more time on the parts that are most interesting to the group.

I will try to structure it so that it makes sense.

I might add some exercises that aren't listed.

I might drop some exercises.

You can ask any question, at any time.

You might need to throw things if I've not noticed that you want more clarity.

You may certainly share your examples – and your counter examples.

Work in groups, share your kit, your ideas and your conclusions. **Share your mistakes**.

You're likely to learn more from each other than you ever could learn from me.

We will use a variety of testing disciplines to shape our testing.

Will will use a variety of conversational disciplines to shape our debriefs.

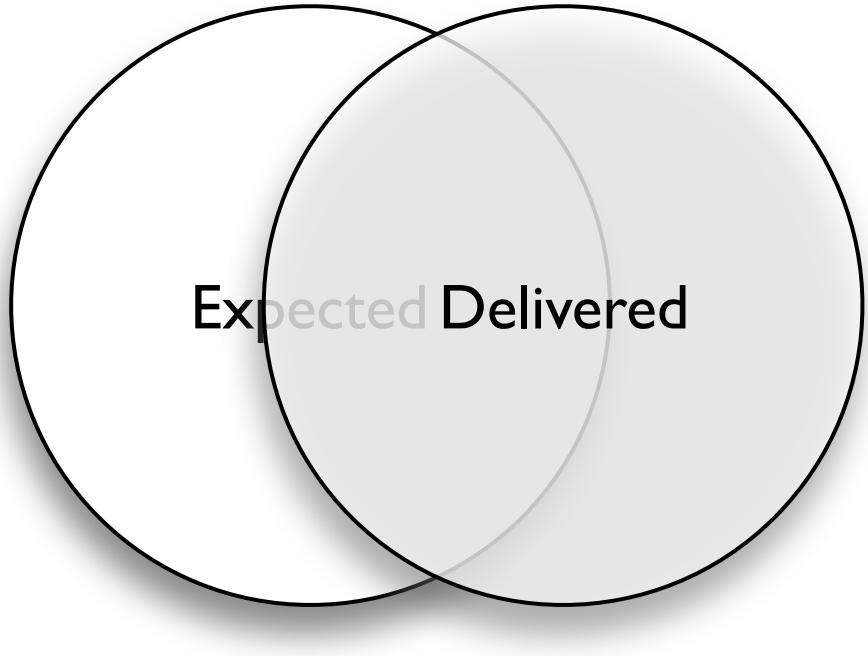
You'll be writing stuff down

and you'll want to find it later. Here are page numbers.

Exploration and Expectation

Checking our expectations tells us whether the system matches or misses our expectations. It tells us about value.

Exploring the deliverable tells us whether the system does something we expect, or does something surprising. It tells us about risk.



It is interesting to explore a working system while consciously avoiding expectations.

The BlackBox Puzzles (generally) are so pointless that it is relatively easy to limit one's expectations. This may make you uncomfortable. So let's try it.

Wicked problems

You may be able to write down all possible combinations of input. All sequences for those combinations. All the elements of your system under test. All they ways they interact. All possible environments. All possible states. All possible data. All the ways you might imagine triggering the system. All observations that you can make with your current tooling. All the known problems. All the known pathologies. Every bit of value that your system has to its makers, users and their customers. Every hindrance it puts in the way of those who wish to harm your organisation, the system, or those it serves. Every edge case and stressful situation it might encounter.

There's a lot of stuff there, much of it useful.

Can you find all possible behaviours? Can you find all possible bugs?

"No" is not a bad answer – it's a **true** answer.

A *wicked problem* is a problem that is *impossible* to solve, rather than *hard*.

- Wicked problems are not understood until resolved.
- Wicked problems have no stopping rule.
- Solutions to wicked problems are not right or wrong.
- Every wicked problem is essentially novel and unique.
- Every solution to a wicked problem is a 'one shot operation.'
- Wicked problems have no given alternative solutions.

(after Conklin)

You can *tame* parts of a wicked problem.

We're tempted to work only with the tamed parts.

So how do you deal with the rest?

Handling a Gamble

- Shorten the odds
- Increase diversity
- Limit losses
- Reduce your minimum stake – easy setup (fast in, fast out)
- Increase your return – usable information is swift, relevant, clear, reliable.

Know that Risk ≠ Uncertainty.

Uncertainty makes us freeze....

Disciplines for Gamblers

- Get better at triggering, better at observing to shorten the odds
- Employ widely-different approaches to catch a broader selection of easy information
- Box time and scope to limit losses
- Use tools to reduce fixed costs and increase scope.
- Build knowledge and empathy to make your information more useful

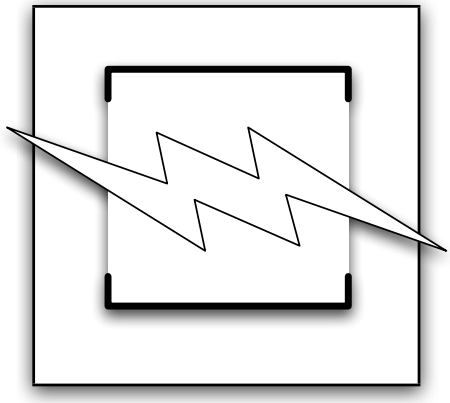
Get comfortable with being temporarily uncertain

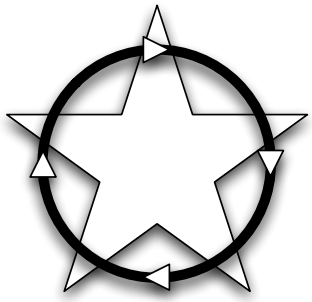
Organisations

- Invest in training, expertise and diversity to improves the odds
- Identify fixed costs that can be reduced with automation – but avoid reducing perception.
- educes the costs? Speed and automation. These can also reduce perception?
- Be able to speedily recognise and reject un-profitable approaches.

Allow temporary uncertainty.

Session-Based Testing



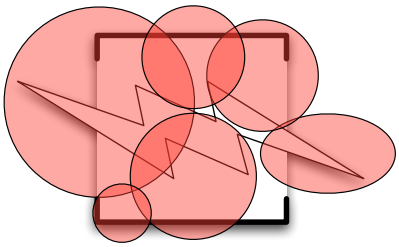


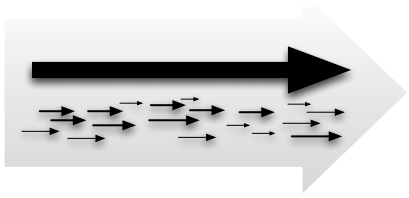
Exploration is:

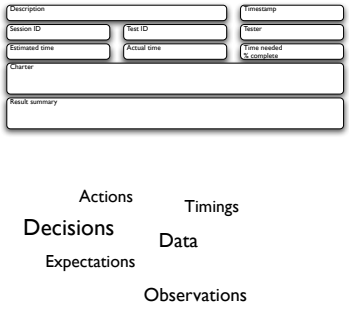
- Open-ended
- Centred on learning

Manage the work with *Session-based Testing*

- Limit scope and time
- Feedback enables learning
- Making good records introduces discipline and enables review







Million Years

You're testing an input field. There's a slider that generates a number in the field, and you can type over the number.

Lots of people have tested this.

Have a look over the input they used.

What common approaches and patterns can you see?

Purpose, Judgement and Discipline

Exploration can be thought of as play guided by *intent**

Exploratory Testing can be thought of as exploration guided by *judgement*

Discipline in test design helps us discover information more swiftly or more truthfully.

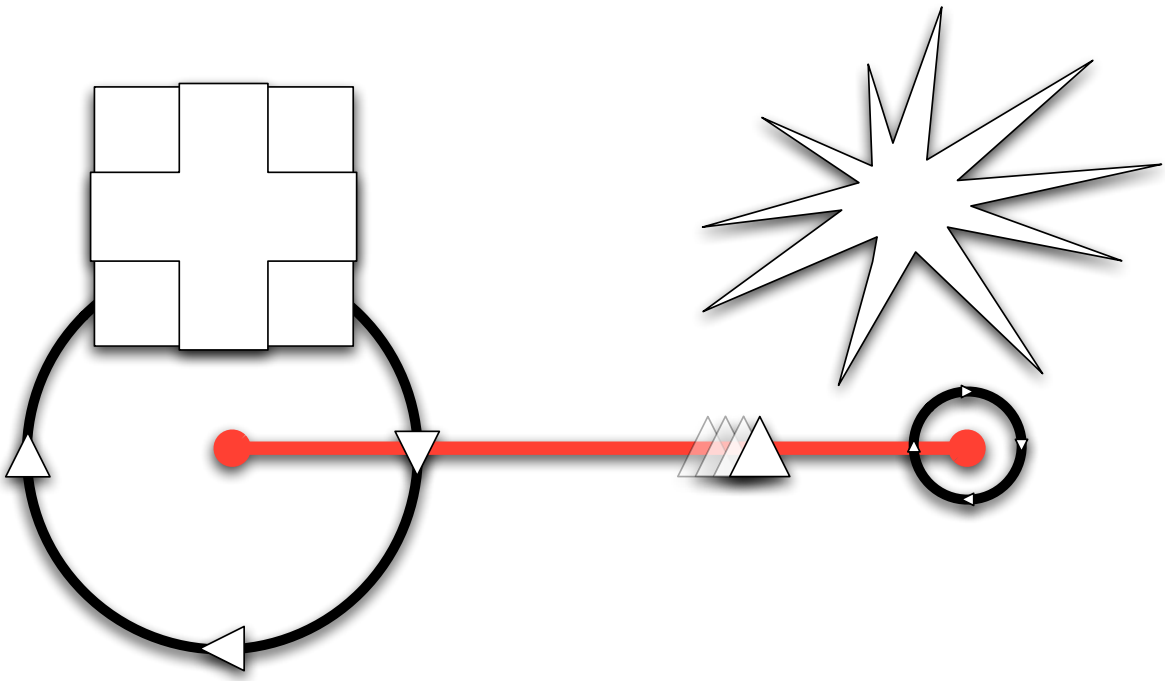
Discipline in test recording enables analysis, learning and trustable evidence.

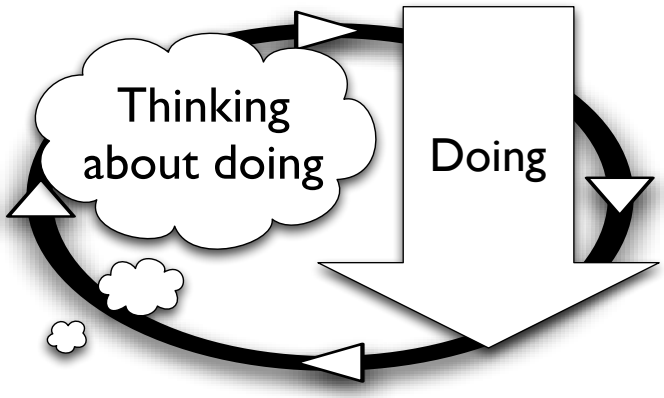
* See Alan Richardson's *Dear EvilTester*, book, chapter "What is exploratory testing?"

Feedback

Some approaches consider the feedback between running tests and making code.

We'll consider feedback between designing tests and using results.





Magic happens here.

Making notes

We keep notes to help our minds stay available in the present, and to support other minds – our colleagues, ourselves in the future.

- Remember – Mnemonic help
- Review – Sharing, improvement
- Return – Historical analysis, long-term project memory

It's key to record ***decisions***. But you might also record...

Identifiers:

- who
- when
- what

Qualities:

- risk
- estimated time
- dependencies

As you go:

- actions
- events
- data
- expectations
- bugs
- plans
- interruptions
- actual time taken
- time wanted
- problems

A key purpose of testing?

We help grow good systems by giving swift, relevant and true feedback.

Swift – to allow control

Relevant – to allow us to use control to steer

True – to steer accurately towards something real

Frameworks for Exploration

We've covered the underlying principles of exploration and how it relates to testing. We've also got an idea of a practical way of working.

In the remainder of this section, we will consider approaches to exploring software. These frameworks help you explore a deliverable for information. They work well with software. Each of these frameworks can be communicated and checked. Each builds a model of the software, and so enable further, more directed exploration.

Different testers exploring the same thing with the same framework might make different decisions and take different steps. However, they are likely to build similar models.

For the purposes of this course, the frameworks are called:

In / Out

Behaviours

Making Sense

Bulk Testing

Mapping

It's good to have the sound on for these exercises, and you'll find that the '*direct*' link opens a larger image than the '*in page*' link.

Framework: In / Out

The In / Out framework is a discipline to help you explore and identify inputs and outputs. It's up to you what you might call an input, or an output. Some people use objects, some people use verbs, some people use verb-object, or verb-object adjective. Some people draw pictures, or highlight a screenshot. Whatever you use, try to group and generalise. As you identify more and more, try to extend the groups you have identified, and include more items if you can. Think about things you might not have included in your groups. Consider new groups, or alternative ways of classifying. Look at the ways you differentiate between members of each group. You'll grow long lists.

When you've got lists, try to identify those items which are linked. Think about how the links work, how you can label, generalise, extend and differentiate them.

This framework helps you to focus on data and transformation, and to refine your beliefs about the boundaries of the system-under-test.

If you find yourself considering the function or use of the thing you are exploring, be aware that you can choose to return to the disciplined framework.

Machine A – In / Out

Have a look at the charter

Start with inputs and outputs

We'll pause and review before we turn to linkages

Discussion

What characterises an input?

What characterises an output?

What kinds of linkage could there be?

Framework: Behaviours

The Behaviour framework is a discipline to help you explore and identify the different ways that your subject works. You'll identify behaviours and events. Behaviours tend to last until an event. Events punctuate behaviours. What persists? What changes? What marks the change?

It's up to you what you might call an event, or a behaviour. You'll probably need to write a few things down before you start to group them. Use your exploration to help you identify and differentiate the items in your lists more clearly.

Once you have a collection of behaviours and events, try to identify and isolate the relationships between different behaviours, and the ways that the events can mark the transition. Can some groups be isolated? Do some seem more important than others? Allow your exploration to guide you to answers to these questions.

Machine B – Events / Behaviours

Have a look at the charter.

Start with inputs and outputs.

We'll pause and review. You may find it helps to draw pictures. Don't worry about scrapping one and re-drawing; it's part of the learning process.

Discussion and triggers

What is an event?

What events have you seen? Have you triggered these events?

What is a behaviour?

Does the machine behave the same way all the time? What different behaviours does it exhibit?

What information is being represented?

Framework: Bulk Tests

Exploring using many dull measurements

Given a possible range of inputs – or interacting ranges – why go one-by-one?

If you can find a way to apply many similar triggers and collate the resulting behaviours, you can design a collection of dull measurements.

Some designs and example data:

• Every integer in a range...	1, 2, 3, 4, 5, 6, 7, 8, 9, 10		
• A regular step...	0, 10, 20, 30	non-integer	3.13, 3.14, 3.15
• An exponential set...	1, 10, 100, 1000		
• A random selection...	22, 74, 81, 90, 4, 51	sorted	4, 22, 51, 74, 81, 90
• A biased random selection...	5.0, 5.1, 4, 4.8, 4.6, 5.2, 5.6		
• Special values...	-1, 0, 100, 256, 65536		
• Range to text...	Jan, Feb, Mar		

You may be able to apply the data using a user interface – but it's easier to apply data, and to capture output if you've got access to an interface that's built to be driven by data or transactions.

Testing with APIs

An Application Programming Interface lets you send data to a system, bypassing the UI. Plenty of web applications have APIs. De-coupling various UIs from a shared back-end processing system is typically done with APIs – and tools to interact with web-facing APIs are getting more common.

You'll typically need some sort of authorisation, the right headers, a request type, and data. You'll get back some sort of response.

Tools like cURL will do this from the command-line, but GUI tools such as Postman and Paw provide wrappers that make life simpler (at the expense of some flexibility).

Frequency

There's a tool to generate data from a range and step – can you do the same with excel?

Try 400-800...

Where next?

Framework: Making Sense

Software is an artefact, a made thing.

Most software exists because somebody thinks they need it.

We try to meet that need by making systems around software.

We know that demand comes from those who pay for, use and value the resulting system – but those people don't value our *software*. They value what it does, the collection of behaviours of the system. This collection is an emergent property of the made thing, not a made thing of itself.

We also note that a system is also shaped by the needs of its makers and their technologies, by the tactics of its exploiters, by the shape and form of its data, by the environment in which the software exists to provide the basis of a valuable system.

When we explore the *actual* behaviours of a system, we see all these – and more – together. We may find problems where the different drivers and shapers work against each other.

In this framework, we manipulate a system and observe its behaviours. We consider whether those observed behaviours make sense in the context in which the system is valuable. We consider whether those behaviours might instead reflect the other demands on a system.

We try to make sense of what the deliverable does.

Machine Q – Limits and variation

Each of these toys represents the same model of rework. You can influence parameters, and are shown the consequences of your choices on a graph.

Controls

- a team has an overall budget, and a capacity for work. (Top two sliders)
- doing work reduces the available budget. Some work needs to be redone. (Next two)
- redoing work reduces the available budget. Some rework needs to be redone. (Bottom two)

Graphs

- Work (or rework) done that does not need to be redone is valuable. The green and red lines represent value already made, and value that is awaiting rework.
- Time increases linearly from left to right.
- There is no representation of budget spent.

If this was a model of software development, it would assume perfect testing at nil cost. We would know infallibly, immediately and for free when work is valuable, or has to be redone.

It's clear that if you reduce the chance of rework to 0, the model gets simpler as one no longer has to consider rework.

What other limits might simplify the model – or drive it to extremes?

Are any limits *variable* ie dependent on other settings?

Decide on what to play with; what to keep constant, what to change. Play with everything if you like, but make the decision consciously.

Play – and while playing, consciously interpret the behaviours you see to help improve your understanding of the model. At what moments were you surprised by the behaviour? What did you do about that surprise?

Know that these four representations all attempt to show the same model in the same way. Their behaviours should be the same, but they are different. At least three of them have bugs. Judge those bugs, and consider what might be going on.

Framework: Making a Map

Exploring and learning *is* making a map.

A map can tell you where you've been , expose where you have to go, gives you an artefact that can be shared.

In this framework, we'll decide on a fixed *base point*.

We'll decide on a *step* – a single repeatable action that can be taken in different ways.

We'll decide what those different ways might be. If they were directions, they might be forwards, backwards, left, right.

Starting from the *base point*, we'll work through all the directions, taking **one** *step* in that direction, seeing what's there, and returning to the *base point* before taking the next.

That's the method. But you can write down a map in many different ways.

The choice is yours...

Making a Map - Machine H

What is a good starting point? Why?

What are the likely candidates for “one step”.

Which is the best choice for you? Why?

Description		Timestamp
Session ID	Test ID	Tester
Estimated time	Actual time	Time needed % complete
Charter		
Result summary		

What kind of map are you making?

When testing, what can play the part of a direction?

Types of notation can include

- Hierarchies
- Flow diagram
- Relationship model
- Directional model
- Trees

Direction matters

Distance matters

Order matters

Mapping

Map-making is a systematic activity that requires – and is driven by – notation.

You choose your starting point, the form of map, your progress, and *where you stop*.

Heuristics, theories and models will reveal themselves to you as you make your map.

Discussion: Systematic Exploration

What makes a systematic approach?

What approaches do you have? What do you use occasionally, or in unusual circumstances? Are there analogies that come to mind?

What makes a good approach? A bad one?

How and When to use ET

What specific tasks suit ET's strengths?

Checking a risk model – Improving unit testing – Diagnosis of beta-test bugs – Emergent behaviours in integration – Getting out of a rut – Discovering what needs to be tested – Looking for security exploits – Experiments in performance testing .....

Overall strategy

Core practice (scripted when necessary) – Bug hunting (reactive, diagnostic, path-finding) – Exploration time (regular gamble) – Blitz (focussed, brief).....

Where in the Project?

Pre-delivery – preparation, toolkit, scope

Ongoing delivery – testing, learning

Approach to live – emergent behaviours

Beta + live – diagnostics

Common Pitfalls

System not ready to be tested

Faulty (ET copes but may not reveal systematic failure) – ET just because there's no documentation / no time – Functionality masked by existing problems

Often forgotten:

Learning process – Tool use – Discipline

Hard to make ET work when:

Spread too thin – Not directed / controlled – Bugs are the only record

Easily-broken Expectations

Every possible permutation – Only bugs and pass/fail need to be logged – Continuing with an unproductive approach

Mind the edges....

Too many bugs! – Reviews take time – Fun

There are Many Ways to Manage Exploratory Testing

(from the management workshop and a currently-on-hiatus blog series)

Common approaches

These seem to arise naturally

Stealth Job

Unacknowledged, unbudgeted, heroic, deniable

Bug Hunt

Enthusiastic crowd, little learning

Gambling

Kick off by risking an unknown outcome

Set Aside Time

Next steps after making a budget

Off-Piste (Iron Script vs Marshmallow Script)

You've worked so hard to make them

Traditional Retread

Managing discovery work with construction metrics

Script-Substitute

Run out of time? Surely ET is the answer!

Acknowledged approaches

Punditry made real™

Session-Based Test Management

Classic

Questioning

Jon Bach frames charters for discovery

Thread-Based

James and Jon Bach propose an alternative without timeboxes

Touring

Handy metaphor with disputed ownership

xBTM

Everything tastes better with a MindMap.

What else sounds fun?

Learning from progress

Daily News | Video Reports | The Summariser

Knowing where you've been

Post-Partum Labelling | GPS – set up automated mapping | Cloudy – tags and word-clouds

Organising the effort

Scouting | Kanban | Following Lenflé

Supporting the team

R&D | Testing Guru | The Inquiring Metricator

Models and Judgement

Models simplify the real world. Every model throws *most* of reality away.

Making models is a key skill for analysis. There are many different kinds of model.

We have already made models of transformation, state, and more arbitrary maps.

For testing, we need to verify and refute our models.

We may build models of success, and models of failure.

Judgement:Found vs Expected

Judging bugs involves a comparison - real vs modelled

Return to previous results

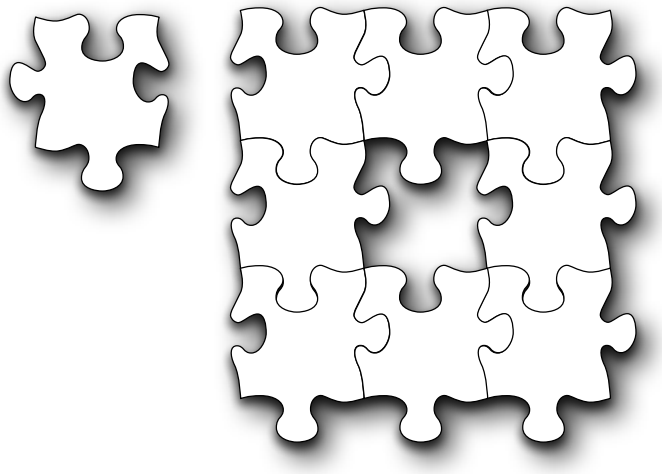
What types of problems were seen by the group?

How did we know they were problems?

A framework for judgement...

- Inconsistencies
  - Internal
  - External - against specific source
  - External - against cultural expectation
- Absences
- Extras

Judgement: Internal inconsistencies



Can we use existing frameworks and testing techniques to observe differences?

How might we judge those differences?

Machine A vs. Machine D

Find the differences.

How are you working through the systems?

Side-by-side?

Against your memory?

Using your previous model?

How are you recognising those differences?

Do you change direction when you suspect a difference, or investigate?

Judge whether those differences are bugs, or are not.

What are you judging against?

Are all inconsistencies made equal?

Can you elaborate your discoveries?

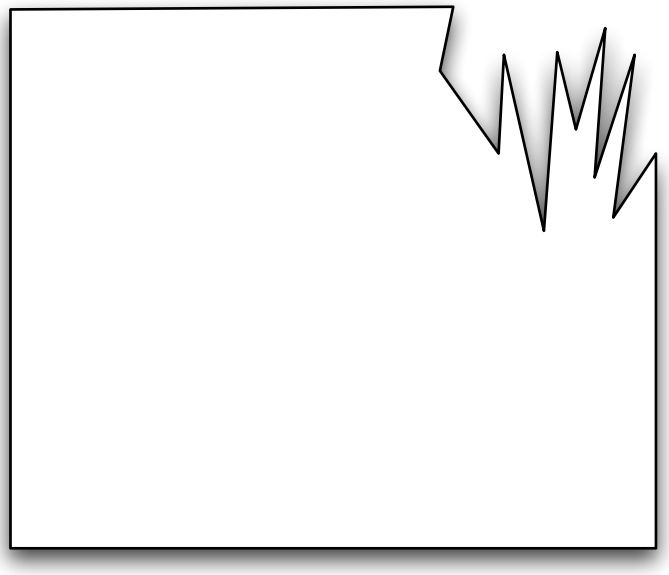
What tests come to mind as you investigate?

Why now?

Could you have used them earlier?

Can you model a difference with a physical analogue?

Judgement:
Inconsistent with cultural expectations



Do we know our cultural expectations well enough to judge against them?

Machine C is a countdown timer

Consider your expectations

What external sources could you check while testing?

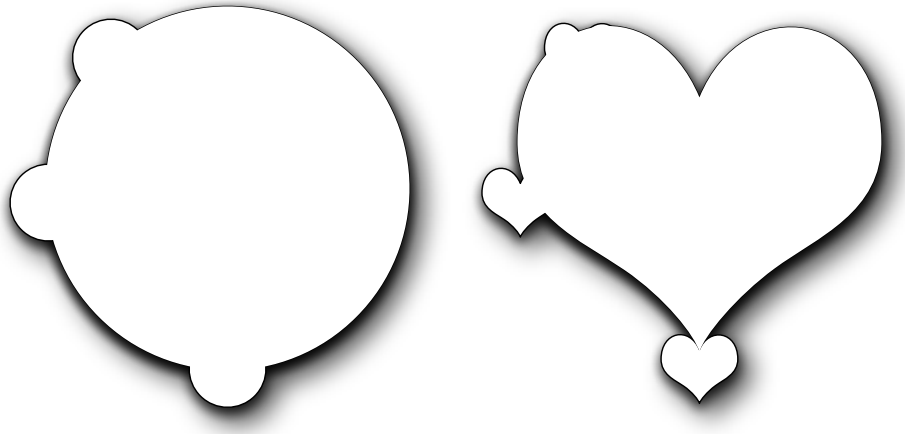
Do you have expectations of potential failures?

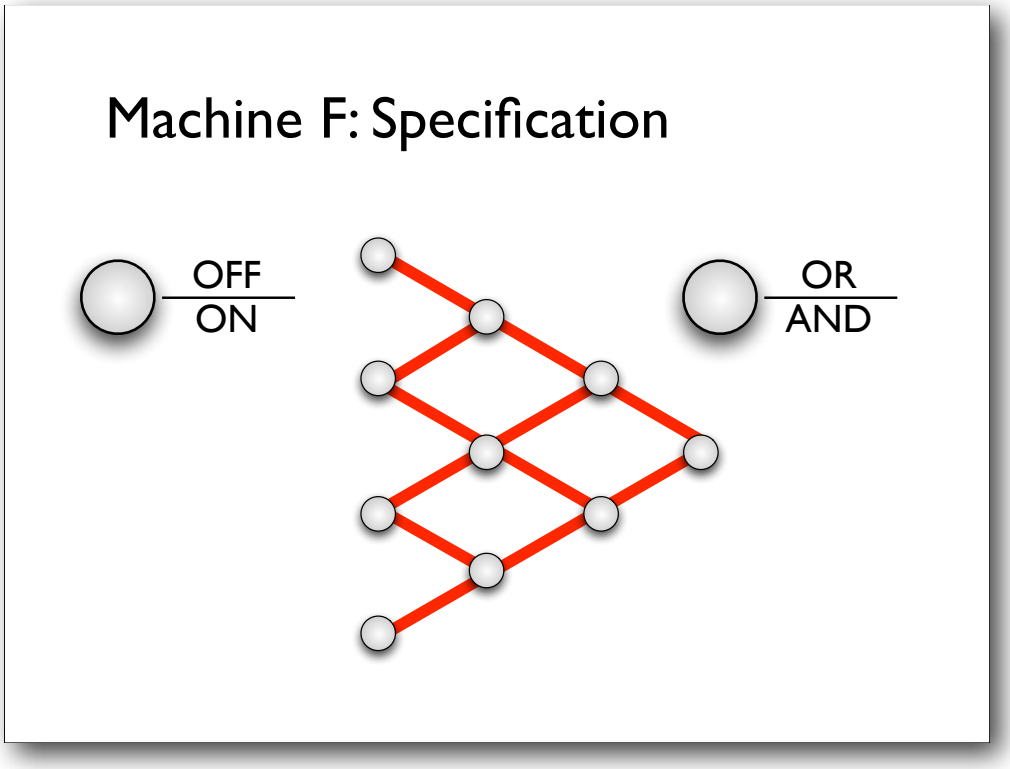
What states might you expect the timer to pass through?

> *Is it worth making a model?*
>
> *Before or after testing?*

How does available time and resource affect your choice of actions?

Judgement: Inconsistent with Specification?





The four green buttons turn their nearby lamps on and off. The six red buttons are logic gates, collectively controlling the state of the lamp on the right. They behave as AND or OR gates. They are linked by the lines.

You have four similar machines. Only one works as specified.

Machine F – code

example 1

```
function translate(input) {
        output = new Array(12);
        if (input[4]) {op_1_1 = (input[1] && input[2])}else{op_1_1 = (input[1] || input[2])};
        if (input[5]) {op_1_2 = (input[2] && input[3])}else{op_1_2 = (input[2] || input[3])};
        if (input[6]) {op_1_3 = (input[3] && input[4])}else{op_1_3 = (input[3] || input[4])};
        if (input[7]) {op_2_1 = (op_1_1 && op_1_2)}    else{op_2_1 = (op_1_1 || op_1_2)};
        if (input[8]) {op_2_2 = (op_2_1 && op_1_3)}    else{op_2_2 = (op_1_2 || op_1_3)};
//
        if (input[9]) {op_3_1 = (op_2_1 && op_2_2)}    else{op_3_1 = (op_2_1 || op_2_2)};
//
        output[0] = op_3_1;
        return output;
}
```

example 2

```
function translate(input) {
        output = new Array(12);
        if (input[4]) {op_1_1 = (input[0] && input[1])}else{op_1_1 = (input[0] || input[1])};
        if (input[5]) {op_1_2 = (input[1] && input[2])}else{op_1_2 = (input[1] || input[2])};
        if (input[6]) {op_1_3 = (input[2] && input[3])}else{op_1_3 = (input[2] || input[3])};
        if (input[7]) {op_2_1 = (op_1_1 && op_1_2)}    else{op_2_1 = (op_1_1 || op_1_2)};
        if (input[8]) {op_2_2 = (op_2_1 && op_1_3)}    else{op_2_2 = (op_1_2 || op_1_3)};
//
        if (input[9]) {op_3_1 = (op_2_1 && op_2_2)}    else{op_3_1 = (op_2_1 || op_2_2)};
//
        output[0] = op_3_1;
        return output;
}
```

example 3

```
function translate(input) {
        output = new Array(12);
        if (input[4]) {op_1_1 = (input[0] && input[1])}else{op_1_1 = (input[0] || input[1])};
        if (input[5]) {op_1_2 = (input[1] && input[2])}else{op_1_2 = (input[1] || input[2])};
        if (input[6]) {op_1_3 = (input[2] && input[3])}else{op_1_3 = (input[2] || input[3])};
//
        if (input[7]) {op_2_1 = (op_1_1 && op_1_2)}    else{op_2_1 = (op_1_1 || op_1_2)};
        if (input[8]) {op_2_2 = (op_1_2 && op_1_3)}    else{op_2_2 = (op_1_2 || op_1_3)};
        if (input[9]) {op_3_1 = (op_2_1 && op_2_2)}    else{op_2_1 = (op_2_1 || op_2_2)};
//
        output[0] = op_3_1;
        return output;
}
```

example 4

```
function translate(input) {
        output = new Array(12);
        if (input[4]) {op_1_1 = (input[0] && input[1])}else{op_1_1 = (input[0] || input[1])};
        if (input[5]) {op_1_2 = (input[1] && input[2])}else{op_1_2 = (input[1] || input[2])};
        if (input[6]) {op_1_3 = (input[2] && input[3])}else{op_1_3 = (input[2] || input[3])};
//
        if (input[7]) {op_2_1 = (op_1_1 && op_1_2)}    else{op_2_1 = (op_1_1 || op_1_2)};
        if (input[8]) {op_2_2 = (op_1_2 && op_1_3)}    else{op_2_2 = (op_1_2 || op_1_3)};
//
        if (input[9]) {op_3_1 = (op_2_1 && op_2_2)}    else{op_3_1 = (op_2_1 || op_2_2)};
//
        output[0] = op_3_1;
        return output;
}
```


Go Exploring

Exploring boundaries and ranges with a binary search

Contiguous range - discrete behaviours

Find two values with different behaviours

Pick the *middle* and find behaviour

What do you know now?

Where's the most interesting place to look next?

Iterate . . .

Machine E

Exploring age validation:

What range / ranges of input?

What output?

Any bugs?

Explore functionality - you could use In / Out / Linkage

References

You'll find clickable references to books and blogs on the workshop site.

The **Heuristic Cheat Sheet** is mostly Elisabeth Hendrickson's, and used with my thanks and her permission.

Use the space below to help you remember interesting references, blogs or books mentioned by others in the workshop.

Quality Tree Software

Test Heuristics Cheat Sheet
Heuristics & Frameworks

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

www.qualitytree.com | www.workroom-productions.com

test Obsessed

Quality Tree Software

Test Heuristics Cheat Sheet
Data Type Attacks & Web Tests

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** (àáâãäåçèéêëìíîðñòôõöö, etc.) ▪ Asian Chars (探索) ▪ **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; *see also Data Type Attacks*) ▪ **Multiple Browser Instances Open**

Input

See also Data Type Attacks ▪ **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

www.qualitytree.com | www.workroom-productions.com

test Obsessed

Debriefing choices

Share insights

Individually decide on three lessons learned. Share them with each other.

Individually, pick one or two lessons from other people that gave you unexpected insights. Talk about those with the group. Recognise new insights, refine existing ones.

As a group, prepare to give two insights to the workshop.

Timeline

In your group, draw a timeline of your testing. Try to show how active you were, the moments where you made discoveries and took decisions, your focus and how it changed, the different ways your group worked together.

Auditor

Pick someone in your group to act as an auditor. Support them if they ask for your help in taking that role. That person will leave you temporarily to audit another group.

When someone from another group arrives to audit you, be nice. Answer honestly.

When your auditor returns, compare your experiences.

Ambassadors

Gather your notes, and make an outline for a brief verbal report. Rehearse the report – try to keep it under 3 minutes. Two of you will leave to deliver it to other groups.

When two people from other groups arrive, welcome them. Listen carefully to their reports. Ask questions (ambassadors too).

When your ambassadors return, compare your experiences.

Watch the testing

Individually, look back over your testing notes and the records kept by the machine. Recognise any places where the records make you change or question your memory.

Take turns to describe your testing, illustrating your points with your notes and with the records kept by the machine. You don't need to use a sequential story: do try a different approach.

As a group, discuss the the approaches you've taken, exploring diversity and similarity.

Collective report

Work together to prepare a report on the things you know about the subject under test. Pay special attention to new and unexpected knowledge – and to how you gained that information. Prepare examples and illustrations. Decide what would be most interesting to do if you had another 30 minutes to test.

One or two of you should prepare to give your report to the whole workshop. The others should support their preparation.

Disciplines to try

Try at least three of these approaches over the day. Change and combine them if you like.

Talk lots, influence each other, switch roles, suggest new approaches and alternatives.

Use whatever tools you imagine would give greatest insight into your subject.

We will share time limits. Remember how long you've got.

Keep notes.

Discovery focus

Two or more **Scouts** explore the subject, positioning themselves to be seen by the observer and reporter. One **Observer** will watch their testing and highlight interesting things, and one **Diagnostician** will dig into surprises. One **Reporter** will keep notes and ask for evidence.

Fact focus

Only one **Scout** will explore the subject, positioning themselves to be seen by the observers and reporters. Two or more **Observer/Diagnosticians** will watch their testing, and will highlight, reproduce, diagnose and keep track of interesting surprises. Two or more **Reporters** will keep evidence, notes of the testing and of the decisions made.

Pairs

Pair up.

Manager

One **Manager** will set out their ideas and priorities for testing. Everyone else tests, paying as much or as little attention as necessary. The manager can try to **steer the group's interest** as the group learns about the target.

Everybody Test!

Everybody is responsible for their own testing – **scouting**, **observing**, **diagnosing**, **reproducing**, **reporting**, managing their **time**, **group role** and **focus**.

Diversity focus

Discuss your test approaches. Each of you should try to **aim you own focus to give the group as a whole the broadest spectrum of interaction** with the subject. As you test, stay informed about the approaches your colleagues are taking, and choose to work together or diversify further.

Tools, tricks and techniques

Write your tool names on pieces of paper and put them in the middle of your table (tools can include tricks and techniques and so on). When you use one or more of those tools, claim your piece(s) of paper. Replace the note when you stop using the tool. **If anyone from your group wants to know more about how you do your thing, they can follow the paper and you can show them.**