**Chapter 03 – Exploratory Testing in DevOps**

1. **Exploratory Testing Overview**

Over past decades, test case-based testing has been critical for Software Engineering. Most software development projects have been writing test cases from upfront requirements. Tests developed to cover as many as possible use cases and the processes that support them. More numerous and complex requirements lead to more test cases. Although test case-based testing has proven itself is integral to Software Quality, however, over time when software development process is speeding up by leveraging more and more automation into it. Then, test case-based testing becomes heavily and hard to adapt with this new change. Instead, it is needing a transformation in its approach. Testing teams are looking to reduce time for writing test cases and looking for alternative solutions and tools to generate automatically most of test cases. Blueprint’s capability to auto-generate test cases based on functional requirements is a robust, here-and-now capability. In addition, test case-based believes the more tests covering scenarios in software use case and requirements, the more quality the software gains. Unfortunately, many bugs happen in ways different from requirements. A software is used in very different context from use cases and in ways we cannot imagine. Recently, Exploratory testing is found as a suitable solution dealing with these problems because it does not encourage creating written test cases, rather exploratory testing is often preferred. The concept about software testing is very different. While test case-based testing puts a great deal of focus on checking the specifications and whether they achieve their goal. Exploratory Testing values exploration and experimentation highly in testing. Exploratory Testing is capable of finding bugs, which may not be detected by conventional means, it provides little to none insight into how to actually perform it.

*What is exactly Exploratory Testing definition?*

Exploratory testing is the simultaneous learning, design and execution throughout all heuristics \* and test oracles. It emphasizes the freedom, responsibility and the wisdom (skills, knowledge and care) to optimize the work.

There are two flavors of exploratory

* Known-Unknown: we aware that we don’t know (or don’t know much) of it. We need to explore to uncover risks that threaten the value of the product
* Unknown-Unknown: is all the stuff that we don’t know that we don’t know it. This needs time, care and skills to recognize

This is a hands-on approach where testers are involved in minimum planning and maximum test execution. Don’t think of it as ad-hoc testing, which is wandering about, not exploring. Exploratory Testing is more like approach rather than a technique.

⚫ A heuristic is a fallible method of solving a problem or making a decision.

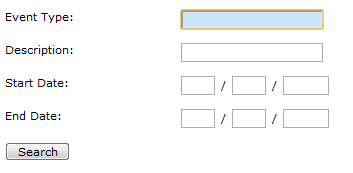
⚫ Two points to remember about Heuristic:

* + something that helps you solve a problem without being a guarantee.
  + that heuristics must be applied sapiently (meaning with skill and care)

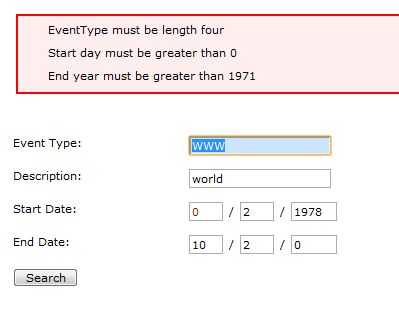
For example: Consistent Heuristic is a heuristic where we respond to a situation in way that allows us to remain consistent. An example: we expect using Ctrl+P as Print Function for all Editors

**Test Oracles**

Oracle has been known as a term that was originated in a 1978 paper by a computer scientist name William Howden and it refers to the source of truth that determines if the output of a system that you currently see is right or wrong. This term is a bit abstract, an example to illustrate Oracle clearly. Let’s look into the form to search for specific events within a given timeline



When we try to enter values below, it turns out an error message



The question is that is it right or wrong? You can say it is right since it tells us that we did something we weren't supposed to. But it could be wrong because it let us pick dates like that in the first place, or because it puts the message down here in this box instead of having a pop-up or something like that. So how can we know if it's right or wrong? That's where an oracle comes to play. An oracle let us answer that question of is it right or wrong

So, an oracle could be something that tells us what is expected in this instance. It can be a specification or the acceptance criteria. An oracle is even an idea from Product Owner who tells us how a product should be. So, in this way, oracle has different forms or source of truth like documentation, product owner’s idea, customer’s opinion, our intuition or comparing to other similar products. But the key important note here is that oracle is not infallible – it means they can be wrong. Using critical & analytical thinking is the best way to use & choose right oracles. We should not limit ourselves in only using one or two oracles, since it’s helpful to use many different oracles to recognize wrong ones and to view the product with different angles.

Oracle comes up with some key benefits as follows:

* To judge whether something is a problem or bug and why we should address it
* To have a better understand how we approach the testing
* To design our better tests. For example: some oracles help us figure out if there is a failure, so they help us figure our what and where we have to look for things

**Test Models**

Model is a simplified representation of whatever aspect of reality we are looking at. When we are looking at a software under test, we can comprehend any part of it, so we will use model to present it in our head. A model appears in explicit or implicit forms. For examples: a map, a software design are explicit models. Otherwise, we get feel surprised when clicking delete button and an item removed without any confirmation, it is a kind of implicit model, because when we are surprised by what the software did, it revealed to us a place where our model of the software differed from the reality of what it did. Sometime this is because the model that we have is wrong, but sometimes it's because there's something wrong with the program that we are testing. When mismatches encountered, figuring out why they are can lead us to interesting insights. Throughout models we can approach our testing with different aspects. For example:

* modeling the system functions by software designs & its architecture or
* modeling user interactions by thinking of problems that users are facing or trying to solve or
* modeling info flow through the application where the data comes into our system, user inputs, APIs, database.. how the data handled and proceeded int the application and then, where does the data go, to a console, a report, a UI or a database..

The fact is that our brain is designed to build on models of the world. We inherently capable of modelling and remodeling our surroundings. Naturally, our brain recalculates the models time by time. For example: in the context of fashion, we quickly imagine how it should be. In testing, models can be categorized into:

* Formal model – is a precise statement of components to be used and the relationships among them. For example: a tax formula, a software design, formal specification in text,..
* Informal model – implies something which lacks of precision. For example: patterns from system failures, product risks,..
* Ad-hoc model – derived at/during/after testing by testers. For example: A tester sees an opportunity to explore a new aspect of system which has not been identified before

The question is: *how to build good test models?*

1. Use project documents: Familiarly, we can build our own models from given specifications, software requirements, user stories, design, development notes, release notes, or even the notes in the meeting with product owners, key stakeholders, developers, or users. These resources are, certainly, helpful because they are describing the product under test. However, we need to avoid biases when using our test models with these resources. Usually, we believe that the product should conform to the output of the developers, or requirements, the designs, so any differences identified are easily treated as bugs. These biases are dangerous and can create inaccurate models. To remove these biases, there are some good practices:
   1. Reach out and gather information as much as possible about products/ features under test from various resources. Each one can describe different facets or different contexts that would help us adjust our models
   2. Find answers for questions like “why ABC is functioning like this?” “Why is it this and not the other?”, “Do these requirements/designs make sense?”, “who/when/where…?”
   3. Share your product knowledge, your understanding with other testers in your team. They may have different thoughts, different models.
2. Use Oracle: An oracle is a heuristic principle or mechanism by which someone might recognize the problem. Oracles are context-dependent. If we have done testing with other similar products, we can use them to compare with current product/ feature. Then, our models are built by our own experiences. we may search oracles from Internet or ask our colleagues who have worked on similar products. Oracle is not only a product, feature or application, it can be something which help us picture the problem, for example: known security bugs.

*How to use models*

There are a plenty of ways to use them that depends on each individual’s experience and their intuition. Someone we have spoken with during the testing seminar at our company, they found useful to start thinking about the models they had and then discovered new ones. Others believed that it works better to leave most of models in the background as implicit things. The models of product will grow and develop over time as they use the product if we put our care on it. As a practice, if we are working on new product, take time to work through and write down all explicit models at the beginning. These help us to understand how the various pieces of software put together and interacted. All these practices have been gathered and structured into a thing that we call the “framework” to use as a guidance for our exploratory testers. The framework is described into the section below

*Exploratory testing is like following a landscape*

Unlikely the way we are doing with traditional approach, Exploratory testing doesn’t use a script to follow. The approach is based on what we have done, where we are at the moment and what we are trying to accomplish as a mission. This is like following a landscape. This means that the actions we are going to do are being guided by the things we see in the application throughout the models we have in our mind. All these things are put together to lead us to what to do next. It’s like an explorer is discovering a new land to decide for establishing a colony. He just disembarked from his ship and has no idea about this new land. The first thing he does is look around himself. He may see a lot of things such as: a forest, hills over here, a river in a distance. Based on his experience about landscape (model) and this land, he decides to explore hills first, taking notes as he goes about features and shapes of the hills. After a while, he sees grassy area where he believes that it is a good place for his colony.

**Heuristics in Exploratory Testing**

“*Heuristic is a falliable method of solving a problem or making a decision*”

- James Bach & Cem Kaner

In the other description, we understand Heuristic as a rule of Thumb, a simple procedure that helps find adequate, though often imperfect, answers to difficult questions. Using Heuristic can help us to

* widen our horizon (give us more test ideas)
* remind us of things we might forget

An example in using Heuristic in our Testing

Undo and Redo are the two operations are tricky to program. Moving back and forth or navigating between pages in a website can cause data loss, data being falsely saved,… So, when executing our tests, try to look for ways or options we can retrace steps and do them, try go back in different ways.

There are so many Heuristics that we can find from Internet. Following is a list of Heuristics Samples

* **Test Heuristics Cheat Sheet for Data Attacks and Web Tests**

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

* **Functional Testing Heuristics**

A screenshot of a cell phone

Description automatically generated

Figure 1 - A portion of Heuristics. Full version can be found at: <http://www.testingreferences.com/functional_testing_heuristics.php>

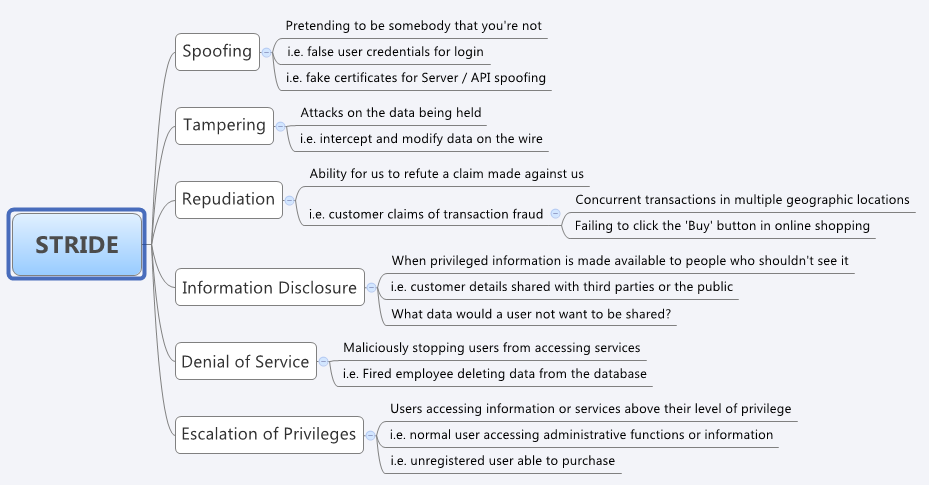
* **37 Sources for Test Ideas**

**A screenshot of a newspaper

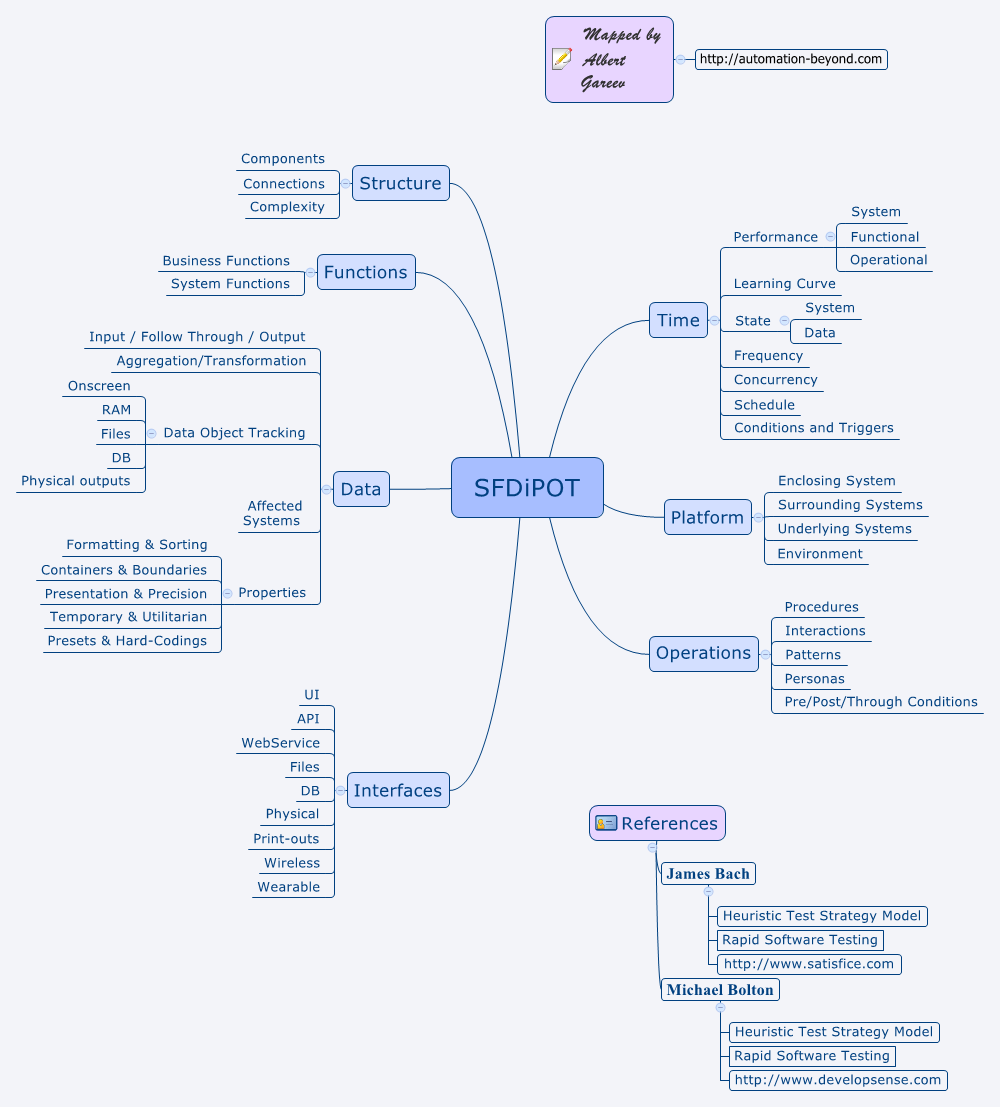
Description automatically generated**

Figure 2 - A portion of Heuristics. Full version can be found at: <http://thetesteye.com/posters/TheTestEye_SourcesForTestIdeas.pdf>

* **STRIDE mnemonic for security testing as the following mind-map**



* **SFDIPOT – Structure, Function, Data, Interface, Operation and Time**



When started exploring the system, we have a general picture of it. We understand the context in which it lives, the kind of data it manipulates, the actions we can take. We may have some ideas about sequences and configurations that might be interesting

Exploratory testing with google earth

Let’s see an example of how we are exploring Hotkeys operations with Sticky Notes

Each tester has their own different approaches. In this example we would start our test session with having an overview picture of what Hotkeys in Sticky Notes. To gain the knowledge, we try to search all hotkeys available for Sticky-Notes from Internet

* Compare the list of hotkeys with other editors to see what new and what left from Sticky Notes
  + For new ones – how’s it working?
  + For left ones – what if I try to use them? Why they are missed from Sticky Notes
* For each Hotkeys try to
  + Follow the instruction to use it
  + Abnormally, use it with differently heuristics such as: many, zero, middle…
* How these Hotkeys work with a big content/ no content?

**A Case Study**

Consider that you are testing a Login page with the “Username”, Password” and “Remember Me” as elements of the page. Easily you are able to turn out a set of cases as follows:

* Combinations of characters, numbers, …
* Combinations of normal and abnormal values
* Maximum size for Input fields
* Empty Values
* Check spaces which are placed at beginning/ middle and end of each individual inputs
* Check & Uncheck “Remember Me”
* And so on….

However, modeling the feature under security implementation, you may want to check:

* SQL Injections
* XSS
* How “Password” encrypted and transferred & stored in database

Sometimes, you also will look at the application cookies to see what happens there. All of these are called as “Test Ideas”. The more test ideas you have, the better testing you carry out.

Various Test models help us have different angles about the feature and enrich our test ideas. Almost testers who are carrying out their testing with traditional approach are boxing their thinking about the product with descriptions stated in the requirements. It makes the testing become poor even the product is really not tested in a real world context. The following practices to generate more test models and new ideas

**Keep watching log files**

Log files are used for developers to diagnose the problems. However, if testers can access to log files to view them in detail – line by line. It would be great deal to generate new test ideas. Log file is a very helpful resource which contains clues for problems.

In addition, testers can compare the log file from this execution with the ones of previous executions for the same test to point out changes from themselves. Some of these changes can be root of causes or turn into new test ideas. While some minor changes may not necessarily manifest immediately, but compound effects can be huge. I have observed the trends of transactions in the log file from an application over different builds. As the result of the analysis, a new process has been triggered in latest builds to process for data replication, but it was not documented in any document.

To monitor the log files effectively, here are some practices:

* *Look into its size and compare it with previous ones*: For example: a log file with normal its size is at 30KB, with active file, it has dumped 1M, so it is of course the thing we have to take a look at
* *Compare the contents of active file and previous ones*: If you run the same test, the same workflow, you expect that the files generated into different runs should be similar. If no, you must know why. There are a lots of free tools support this comparison and highlight differences.
* *Pay attention on date/time logged for major workflows in different runs*

**Keep watching competitors or similar products**

Most testers only focus on the zone that they control, on features their developers have built, or what they are asked to test. However, if you take your time to see what the competitor’s products are, how they are built, what their issues are, risks or known bugs. This helps you develop your test ideas better. Key scenarios probably translate well across the teams and products. For example: if you test Lazada’s payment function (online shopping website which is rather popular in Asia in general or in VN in particular), let’s look at how similar function works with Amazon, or security mechanism used in Amazon can be compared with Lazada.

All feedback, comments, bug reports from users who are using your competitor’s products will very valuable to your testing. These are feeding your thinking to get more insightful test ideas.

**Pay attention on failure modes**

Any failures are good models to learn. These models help us to build more of test ideas when delivering the testing. By answering “what if” questions will trigger your thought for new test ideas. For example: Assume you are testing a Login function of a web application, addition to “normal” cases, you can ask:

* What if an authenticated user tries to login the system?
* What if an authenticated user closes the application without logout and try to log after then?
* What if a user tries to copy & paste into password field?
* What if a user wants to show his password during typing?
* What if a user press and hold (many clicks) “Login” Button at once?
* What if a user operates this function with other web browsers?
* What if an attacker tries to perform dictionary attack?
* What if a user blocked from entering wrong username or password more than 5 times tries to login with a correct account
* What if a user remembers his account info, then how its cookies managed?
* …

**User’s emotions**

Happy cases, alternative cases and exceptional cases are terms frequently used in designing test cases. However, if you stand at user’s view, you can see that user’s emotions can involve into generating test ideas. For example: there is not only happy case, other emotions can be considered such as: scary, angry, embarrassing, desolate, forgetful, greedy, stressful. Each emotion constructs a set of different activities which can harm your application or turn your application into context which cannot be control. For example: when a user feels angry, s/he can operate the application in context of out-of-control such as: shutdown the application suddenly even when the application transaction is in-progress, or perform crazy clicks on the whole application.

A set of emotions is a heuristic that helps the team to design more tests during exploratory execution. To make this work, you can think about happy path and look along it for alternatives. When going through the happy path, keep thinking & seeking for other paths that could be taken using this set of emotions.

**Functionality Testing is Good, Capability Testing is Better**

User stories which are implemented into features are a solid starting point for coming up with deterministic check. Usually, testing teams end up proving what they have seen in story descriptions.

Product is a solution, but if the problem cannot be solved, the product doesn’t work. Testing the capability is to verify the solution with given business problem. In addition, focusing on capability leads to deeper insights and prevents tunnel vision. To test the product capability, you have to put yourselves into real-world context where the product can solve given problem. For example: A Helpdesk Feature allows users to contact technical teams for their troubles, Helpdesk creates a ticket to submit to our teams. Under the view of capability, the system is not only to function correctly to create tickets, but it has to offers an alternative channel to deal with situations when the network access unstable or our web services get shutdown by any reason. For these situations, we expect to an email prominently on the form.

1. **Exploratory Test Management**

One of the challenges that testers confronting with is to demonstrate how the testing they have done. Traditionally, to do this, testers use the classic test scripts. These documents are listing out a list of steps associated with results that they expect to see for each step. Of course, approaching the testing with this way, testers will lose creativity and learning if they overscript the tests.

So, how can we stay on the other of spectrum and still have a structure that we might need in more regulated environment such as insurance, banking, or healthcare? Session-based test management is answering the question correctly. It allows us to structure our exploratory testing without needing have each and every step prescribed. This concept was initialized by James and Jon Bach. It allows us still have discovery and creativity, but it does so in a way that lets us setup a level of rigor and accountability that we need for the context that we are working in.

There are some couple properties of session-based test management comprise of an effective exploratory testing

* Charter
* Time-box
* Session Notes

**Test Charter:** A goal-oriented description for exploratory testing. It is a part of session-based test management that precisely and concisely guides testers what they are looking for in the test session.

Each test charter has its specific goals and contains a specific portion of what needs to be tested. A charter can be a UAT Test for a story, a screen, a bug, a business transaction, or a technology facing for a piece of feature. An example of test charter:

*“Explore unexpected failures with power and network interruptions to discover if the product recovers gracefully”*

**Time-box:** There is no standard to regulate how much time we need spend on our exploratory testing; we can use any size we want. However, if we take too little time for our exploration on a feature, we will have nothing new to discover. Otherwise, too much time to do will use all energy that results missing a lot of things on the way we test. A practice recommends using a time-box between 45’ and 90’ for each charter execution or called as “test session”. During a session being taken, testers are strongly recommended turning off all notifications which can distract them. Focused testing time like this will yield enormous benefits. and we need focused time to be able to do it well.

**Session Notes:** Taking notes what we have been observing and our thoughts, what we have done is crucial. All these notes are benefits to review after a session completion. They remind us of what have just happened in the session, then are useful to improve our testing in next times.

However, taking notes for exploratory testing is never a simple task. If we spend too much time on taking everything during testing, we won’t acquire much knowledge about the product. Otherwise, too little things to be noted will cause hard to review later on, even we don’t have enough test proof to demonstrate what our testing has done in regulated environments. Therefore, testers need evaluate, select and use wisely different tools as well as deciding what to take notes during a session. There are a plenty of choices from handy note-taking such as: Notepad, WordPad, Rapid Reporter,… to automatic recording tool such as: Screencastify, QMetry, One2Explore (we will describe more details about this tool this section),… These tools bring great power for testers in logging their executions and test data.

1. **How Exploratory Testing works in DevOps Culture**

While traditional testing finds itself as a blocker that is slowing down the process of software delivery or even being worse it is stop the process when a failure found. Exploratory Testing is being considered as a right approach for Agile & DevOps manner. Why this is happening? To answer the question, we need take a look on the spirit of DevOps and how Exploratory Testing aligning with it

DevOps is more directive than Agile. It stresses collaborations and integration between software developers, operation personnel and everyone involved into the design, creation, development to regularly create pieces of useful software. Or say in the other way, DevOps is to combine all those associated with delivering a software product into a unified, self-directed, and self-correcting team that follows agile as its work process. The teams in DevOps project intend to deliver small features incrementally and iteratively, starting with a minimum viable product, a release used to learn from production use.

In this manner, to fit in this environment testing needs to have great insights into how product used, having a great collaboration with other teams to provide quick feedback and not to lead time in making long-term plans, instead spend effort for test execution. Exploratory testing is demonstrating itself as a best fit into it because

* One of 7 context-driven testing principles that exploratory testing adapting – is “*People, working together, are the most important part of any project’s context*”, it totally fits with spirit of team collaboration in DevOps
* It uses test models, heuristics, oracles to carry the testing in real-world context. Information from operation help tester have good models, otherwise, their testing is laser focusing on real context in operations to maximum eliminate faults from it
* It approaches the testing by evaluating the product by learning about it through exploration and experimentation, which includes to some degree: questioning, study, modeling, observation and inference, including. This is exactly what DevOps working, they create small software pieces to get quick feedback on the software as a way to learn how the software brings the values for users
* It uses visual methods such as mindmap to explain testing ideas and increase efficiency of team collaborations.

**Value of Exploratory Testing in DevOps**

*Rapid feedback*

One of key tenant in DevOps Environment is rapid feedback, it works in short iterations so that we can get faster feedback and it tries to constantly produce working software, then we can get immediate feedback on it, even when it is in production. Quick feedback loops help learning our product faster, then we can adjust what we’re doing. This is exactly what the spirit of exploratory testing is promoting. Testing in DevOps is supposed to be included as a part of each iteration. It differs from traditional approach where testing often occurs in a testing phase at the end as we neared release time. In DevOps manner, testers will get involved into a lot of activities as follows:

* Working with developers and product manager & others to figure out details what exactly it is we are going to test
* Doing research to figure out how client interacts with a new feature by from data collected from our company
* Having a discussion with the developer on how this feature fits into the overall application and what things it may rely on and what data it might use and produce.
* Testing prototypes to see how it’s working to give developers quick feedback.
* Working with developers to identify what we need in place to make it easy to automate regression tests

By all we said, exploratory testing is not only using the product, but it’s also experimenting & learning and helping other to improve the code. It can be executed at any stage in a development process

Exploratory testing in DevOps is leading testers to an interesting thing to perform it in production. Most of us think about testing as activities taken before the product released. The idea of testing in production kind of flips that idea on its head a bit. It can take the form of actually trying and using the product on production sites to find out issues. It is not often what’s meant by the concept of testing in production. Usually when we talk about it, we're talking about things like observability and using telemetry and monitoring to see what's going on at the production site. Or we're talking about things like A/B testing, or other concepts of that sort. Exploratory testing is really fit into this environment. Testing on production is doing experiments with different things to see what happens

A big company offers children games & videos has hired testing teams (in form of test-sourcing) to carry out the testing on their production. They built a checklist that can be referred as test charters for the team to explore their services from the site. Key items defined clearly in the checklist like: explore the site to find out dead links, explore the site to game/video descriptions not matching with its title or inaccurate, explore the site to see how UI layouts displayed,… however, the checklist is not constrained testers to add more items/ charters that help build in their service quality. Otherwise, some organizations use crowdsourcing to carry out testing on their productions. The testing in this model is performed by a crowd of testers working at different workplace, the process is online and leading to significant results. Testers in this model only get paid when their work get done or the valid bugs found as the first. Because that reason, testers must use all their skills and their time wisely, they rarely develop test cases instead prefer value of exploratory testing to cause problems as fast as possible. Some Crowdsourced Testing Companies are building their success with this model such as: Global App Testing, TestIO, Crowdsourced Testing, uTest,.. Although Crowd-Testing sounds like working well, any organization use this service with their DevOps must deal with a paradox between this model and DevOps methodology. DevOps methodology propagates co-existence of all teams in a joint software development environment. Crowd-Testing against this propaganda as it involves testers from different locations working on a common project. Communicating and coordinating will be extremely time-consuming and costly. Therefore, wiring up an effective communication model among testers and the organization a survival for this collaboration.

*Better finding issues*

Exploratory testing discovers unknown issues during and after the software development process in a way that requires minimal prep work for us allowing us to quickly dive deep into a feature and assess its quality, providing the team with rapid feedback on how to move the feature forward. Exploratory testing plays essential role in determining the overall quality of an application because it serves to surface unexpected or unknown bugs

In this approach, testers must be able to think like the application’s users and participate how it will behave. Exploratory testing may be open ended, it leaves a freedom for us to decide how and what to test on our own. More prescriptive subsets of exploratory testing include strategy-based exploratory testing and scenario-based exploratory testing, both of which require us to focus on specific areas or user flows within the application. When a tester is given a more freedom in their testing, with their wisdom and sense of responsibility he is usually to find out issues faster and discover issues which are hardly identified by a script

*In Operation Support and Continuous Exploration*

* Release Process – Once a feature built and ready for production, it has to actually be released to production servers somehow. This process is a prime place for applying exploratory testing skills. Testers need test things like the speed of release process, the ability to pause it, to stop it, to backtrack it and other things in release process.
* Monitoring – when a system released, DevOps team usually wants to know where things go wrong in production. To do this, an alert system can be established to let us know that there are errors. However, in a complex system (or an ecosystem), there are many errors can be encountered. The question is that which ones are matter with us. Figuring out what kind of errors map to the things that we care about and are concerned about is something that testers can help with
* Finding bugs – reproducing issues is a typical example of helps from testers. For example: testers see an error in the logs. Then try to understand how it happened. So, by looking into the logs, making an educated guess, playing around with the system, eventually testers are able through a process of elimination to reproduce what the customer had done to cause that error. Activities like this are inherently exploratory.
* Operations - Things go wrong on production servers. Databases get corrupted, a server goes offline. Issues happen, and things need to be responded to. So monitoring can help us know about these things, but then we actually need to take steps to correct them. We can improve these steps with the help of exploratory testing

**A Framework Practice to perform exploratory testing**

Every organization has their own implementation for DevOps and decide how their testing integrated into it. This will be explained more details in the **Chapter 07 – DevOps Testing.** Whatever it is, exploratory testing must go through an intellectual process when it is carried out. The process will be different from individuals that depends on their experience and practice. Below is a framework practice for anyone who want to apply it into DevOps

Test Execution and Report

Learning and Exploring

What& how to test

Snapshot the product/ feature

Initiate Test Ideas

The System

Designs

Similar Products

Development Notes

And others

Formal: designs, graphics, stories

Informal: Notes, conversation, ideas

Our Learning and Exploraing ussually go through the following phases:

A test model can be built by many different sources in formal or informal forms. These sources help us imagine and model our product with different views. The more sources we have, the more knowledge we obtained about the product. A list below is key sources can be attempted during exploratory testing taken:

* Other parties: from the discussions, interview, questioning stakeholders, clients, users, developers, Subject Matter Experts…
* Documentation: software specs, designs, requirements, user guide, Release Notes, Development Notes …
* Experience: what we had from similar products or even from the product being under test by your experiments
* Similar product: what we snoop on other similar products

*Obtaining*: Take in the information about the product from different sources which are described above. For instance: when reading, assimilating, analyzing a requirement document, we can ask the author to classify in his way to reveal other product angles or we can ask open questions to get a big picture initially. Or use “Always/Never” Heuristic to question and find out problems that we might be blinded by our lack of inexperience. For example Testing for a banking project, testers would ask stakeholders “what is (never) always happened for a transaction?”. An answer can be “No transaction lost”, then we understand that the product should have a mechanism to monitor & track all transactions, the corrective response is required. Or in a healthcare project, a tester can ask PO/PM “what must always happen in the product?”, an answer could be “PIH data must be protected all the times”. We understand the two of things must be cared are that (1) data loss and (2) data breach. We must identify an approriate approach to verify how the data proceded, stored, transferred, encrypted (if needed), backup… and how data to be classified for restricted accesses

*Modelling*: This is an essential part in test execution. It represents the thought-process that we believe is going on in our mind when we explore and test the product. We build test models from many sources and use them to validate and challenge the sources. A test model helps testers to recognize different aspects of the product under test. Test models can take various forms:

* Checklists: Comprise a list of functional area, features, or issues to cover the testing
* Graphics: A direct diagram that consists of nodes and links that connect them such as: state-transition models, control-flow graphs, flowcharts, sequence diagram
* Business/ User Story: The concept of user stories has wide acceptance in Agile teams. Story summaries or feature titles are usually augmented by acceptance criteria
* Mind-map: decompose the product to smaller ones. They can help people to visualize the structure of the system under test.

*Converging*: Build and Use Oracle is decide for a problem in during exploring and learning the product is important. *“A Test Oracle is a mechanism to determine whether software executed exactly for a test. It enables us to predict the outcome of any test. In effect, an oracle tells us what a system does in all circumstances and situations. If an oracle does this, it is good. Our Oracle might be derived from the same sources as our models. Whatever our sources of knowledge are, like models, they are fallible*.” An example is that we evaluate the capability of OpenOffice by comparing itself with Ms. Office*.* Or when testing “Drag & Drop Picture” of StickyNotes, in Ms. Office, we can drag a picture from an Internet browser and drop it into a document. But this doesn’t happen in StickyNotes, then it is turning back to challenge our Oracle and raise a concern that StickyNotes differs from Ms. Word that may cause violated expectations. When we are modeling, we can derive examples of a system in operation, suggested by our model and use these examples to post ‘what if?’ challenges to our sources

*Challenging*: Questions take into the form of scenarios that we believe that the system must deal with. These confirm our understanding. When there is a difference between a test model and software, we will need know if it was a bug, a failure, an issues came from the software or it was a problem of our test momel. Test models are not always right. When an incorrect model encountered, we have to get back to and validate, challenge and even change the sources.

Test Execution and Reporting includes activities of test experiment design, Perform these experiements and Report what we found.

*Design & Perform Test Experiments*: use models to generate things for our testing (called test ideas or coverage items). These test ideas are written in form of test charters which will be executed with given time-boxes called sessions as mentioned above. Performing Experiments is to enact test procedures to discover reliable answers to our questions about the products. During this performance, we observe outputs and judge whether an output matches an expectation or judges whether the outcome of the test is anomalous in some way. Any abnormality encountered needs to be interpreted to find out causes or have more info for making a decision of demanding or not more other tests. This is kind of diverging process taken in our mind to generate a number of accounts of different experiences from what we have done with the system

* The product behaves like this is exactly the same as the expectation (or not) in just this situation (or may not behave correctly in other). Do we need more tests?
* The product behaves in a way that is different from our model. Do we need to challenge the source and align the test model?
* The product behaves in ways that testers (or tools) were not seen (or recognized). Are testers (or tools) are fallible?
* The system cannot be made to fail in ways that stakeholders are concerned with – perhaps stakeholder concerns are addressed or perhaps our tests are poor
* The product’s behaviors are patterned that gives to new concerns, new risks, and more tests.

Tools can also be applied to capture outputs and make comparisons with the expectations, but keep in mind that tools are not a sophisticated observer, it is just used for basic checking activities. Encoutering these limits, MeU Solutions has just invented a AI-based Testing Tool called One2Explore that helps track/trace and report all testing in pictures/ graphs.

One2Explore is built upon the concept of visulazing all testing facts that concisely & precisely conveys the quality reports as well as improving the testing efficiency. One2Explore offers various features, the two of its key features that benefits for Exploratory Testing are:

* Capturing all user (tester) actions and relevant data for the web and mobile applications and visualzing test execution by graphs
* Analyze test executions, compare current test with other previous test execution by graph algorithms to find out missing scenarios and make better test performance

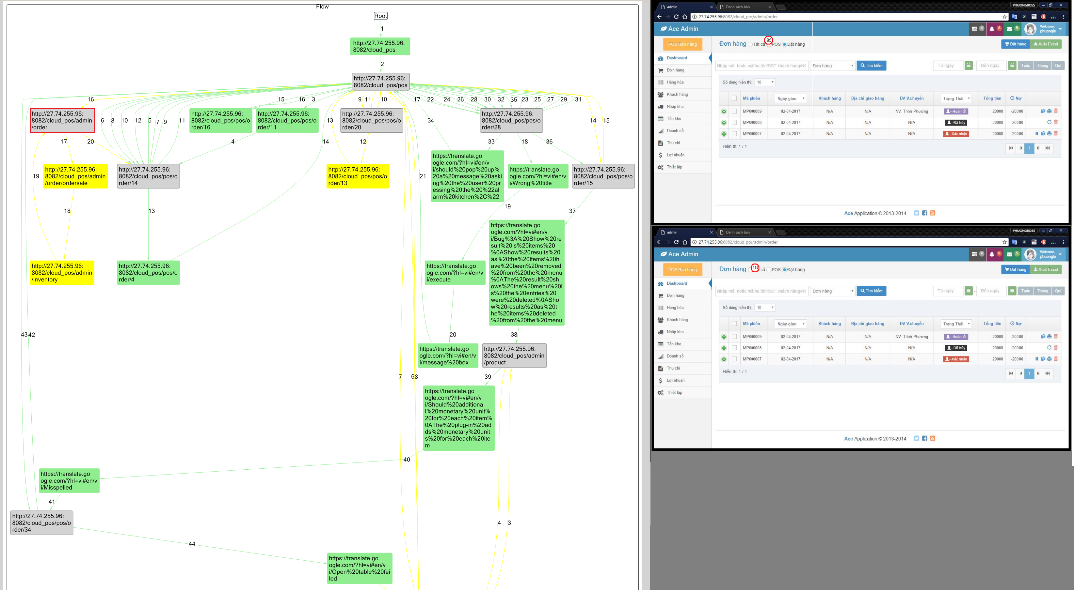


Figure 2 - A Graph represents test executions from One2Explore

Adding One2Explore here for a tool to take note

*Logging and Reporting*: Logging abnormalities (also known as failures, bugs, defects, errors, issues,..) can follow a formal process that involves both many parties (dev, test, user, client,..) and tools (JIRA, Bugzilla, TestLink, …) into series of activities of review, controlling changes, re-test, regression testing,.. or it can be an informal one whereby the tester talks to the developer and agrees on a certain point. In addition, all testing evidence (or test proofs) which helps convince a statement or conclusion and anything useful which improves the testing performance should be noted for the debriefing later. Following is a list of recommendations (but not limited) to be noted in a session

* How the tests performed
* What we did. Why we did
* What we saw. Video/ screenshots
* Questions and New Ideas?
* Our evaluation for the quality of current charter
* Ratio Test: Setup, execution
* Challenges, blocks, …

*A debrief is a discussion around a recently completed exploratory testing session between two people, the:*

* *Reporter – or the person who ran the exploratory testing session*
* *Reviewer – or the person who learns about what happened during the exploratory testing session*

*During the debrief, the reporter shares information such as what they did and didn’t test, what they learned during testing, what issues they faced and what bugs they raised*

Finally, reporting is the process whereby the tester provides meaningful feedback to stakeholders. Usually, it relates to the completion status or coverage of tests to give an indication of progress, but it will also give an indication of completeness or thoroughness. The status of individual tests is of interest, but it is more the patterns that emerge from the interpretations of these tests that inform the decision-making of stakeholders. A good report is meaningful to stakeholders who directly receive it. For example, it is much sense to send a report with data & info about tests which relate to the code coverage, control flow to developers rather than a report contains the ratios of pass/fail or number of hours spent on testing

Example of exploratory testing report

**Skillset in Exploratory Testing**

Going through this thought-process, tester uses all testing skills and tools to achieve their testing goals. There is no limit in the skills, we can list out some key skills which would be helpful

*Persona* Testing

A persona is a fictional character that you create to represent parts of our audience. Persona is a great tool to consciously adopt the habits and feelings of different people. It provides us more different views about our product to discover different type of problems.

There are plenty of factors to determine our software personas. Following are some samples

**Age**: Age decides our behaviors. For example: Teenager user is more technically savvy. Many have been growing up using the Internet, so navigating the web comes naturally to them from social media to online shopping. However, while they may be more equipped at handling complex features, often their standards are higher and patience is thinner. They will not stick around for complicated forms or slow load times but will leave our site for a sleeker user experience. While older people may be less impatient, on the other hand, they also may be less likely to get around more intricate functions. They are using our web application slowly and taking time to read every page, frequently use the 'Back' button to remind them what was stated into previous pages and often seek out online help so assist them.

**Role & Job**: What does the people do impact their behaviors and what are they using it for that will influence their decision? For example: A businessman who usually travel abroad. He has a great need to receive the reports out of business hours, accessing the application from different locations with time zones. He may use a variety of browsers, OS, and devices. All of these are absolutely different from office users who are stick with a regular time and jobs

**Accessibility**: considering different challenges people may have when it comes to user personas can be helpful. For example, many men are prone to color blindness, which means something as simple as a color choice could make or break the experience for someone. Users that are hard of hearing may rely on image attributes and labeled form fields.

**Geolocation**: Looking at demographics is a great way to understand personas, but it’s not just limited to age and gender; it also includes looking at where people are in the world. In fact, where someone’s looking at our site from has a big influence on how they’ll use it. For example: An Asian Users have different habits from US Users. Asian ones may stop at a page for a while to check out almost items there and pick the most they like. US has a clear goal set at the beginning when they start their shopping, so they will go directly to what they need.

An exercise to get you familiar with this testing, there is an application built and used by

* Mary is a technical lead; she is always busy with her calendar. She appears an impatient person to tries to complete her jobs in haste
* Tom is a journalist; he is curious with new things. He tries to find sense of things that he is working with.
* Paul is a businessman; he periodically uses the application to retrieve the information or complete his task

What are test ideas that associate with her persona? (Below are some, but not limited)

|  |  |  |
| --- | --- | --- |
| Mary | Be a Technical Lead; always busy and impatient  Access the application internally;  Use the application frequently | * Tests need associate with quick workflows when she will use them to perform her job * Hotkeys tests needed * Abnormal operations can occur deal to her impatience * Performance is needed in the testing * The application should be easy to use and fast to complete operations * Compatible tests when Mary may use the application with different OS |
| Tom | Be a Journalist; curious person  Access the application internally and externally  Use the application periodically due to his jobs | * A journalist usually pays attention on content that need to associate with tests * Tests with different Internet connectivity, and sometimes the connection is poor * Because of curiosity, Tom will explore all possible workflows/ paths * Access the application with different devices like mobile, laptops and PCs and different OS * Time remains on using the application is short |
| Paul | Be a Businessman; usually travel  Almost access the application outside  Use the application periodically | * The application should be available all the time; its downtime should be short * The application reports should be concise and precise * The application must be accessible outside where its server is not located * UI/UX should be eye-catching |

*Risk-based Testing*

Risk-based Testing is a risk-focused investigation approach in where the effort of testing is distributed on items as results of risk assessment. Risk-based testing is more perceptive than calculative. The objective of risk-based testing is to minimize the risk associated with the product. Exploratory testing combines with risk assessment will bring more efficient. Assessing risks during our exploratory testing will give a clear direction, remove wastes and stay focused on the project’s goal. Throughout a chain of activities including: modeling, learning, designing and performing the experiment, observing, and asking & logical reasoning, risk-based testing participates into this process as an efficient driver to make our testing to stay focused on having more value of our testing

The more risk eliminated; the more confidence of quality gained. However, we must balance between the cost of effort for risk assessment versus the benefits it brings to our testing. To answer the questions “how much time should be allocated for risk assessment is enough for our exploratory testing?”. It totally depends on the context of the project and the product we are working on. Using the following question list as a heuristic that appropriately leading us

* What are the things that end-users (or key stakeholders) think that they should never happen? For Example: it is critical if a transaction is lost from a banking system. Then all charters and risks associated with processing the banking transactions must be placed in high priority
* What if this/that function fails? For example: if the answer is that it is not allowed to fail to cause the whole business interruption. Although this answer is not very sensitive for testing, it helps to understand how important the function expected, then we can plan out for a significant effort of testing to mitigate risks in our exploration.
* What must the product comply with (standards, regulations,…)? For example: HIPPA requires an accuracy in protecting health information held by the covered entity or business associate. It is necessary to have a (or many) charters to conduct an accurate and thorough assessment of the potential risks and vulnerabilities to the confidentiality, integrity, and availability of protected data
* Questions related to level of expectations for performance, reliability, usability, compatibility,… to evoke requirements different kinds of requirements. What if the requirements associated with any of these elements are not met?

Risk assessment must be transparent with our clients and key stakeholders. They may not be aware of how important to manage risk in our testing, however, having them involved into the process is a way to make a great awareness. Risks need to be explained to key stakeholders in a language of their knowledge. For an example: we cannot speak to a business owner risks of technical-related, but they can explain them how it impacts on their business value. A practice for making risk awareness is to organize our testing into risk-associated mindmaps to bring our risk assessment to our stakeholder at each their review

The following is an example of how we - MeU Solutions test team – have been using risk-based exploratory testing

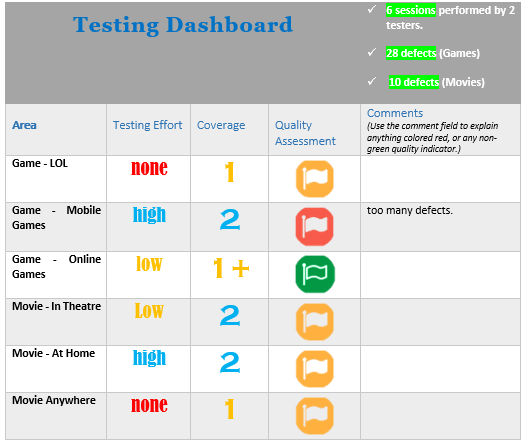
*Risk-based in Test Planning for Exploratory Testing*

Using Session-based Testing Management is a good practice to employ risk assessment in our exploratory testing. This session-based approach brings a clear structure to loosely defined exploratory testing. It is an approach to planning, managing, and controlling Exploratory Testing in short (almost) fixed-length sessions. All test charters are marked with numbers which represent its importance in term of testing value which is assessed by many factors such as: business impacts, technical constraints, technology,… Higher number indicate more value of testing delivered to the product. The sample below shows that charters marked by 1 are highest priority for execution. By doing this, risk assessment communicated regularly with key stakeholders, it also translates test strategy into a visual map



*Risk-based in Test Report for Exploratory Testing*

By results from test execution, testers decide level of risks which associated with each module/ function and placed them into the test report to convey the info for quality risk assessment that helps key stakeholders to make decision for “Go” or “Not-To-Go”



**Workflow Testing**

Workflow Testing is not new. It is really used in a lot of testing with traditional approach. The technique is to route a record through each possible path. These tests are performed to ensure that each workflow process accurately reflects the business process. It consists a set of activities to create a desired outcome. Workflow can be either be in a sequential manner or parallel with multiple steps occurring simultaneously.

**Tour Testing**

If we are visiting a city where we have never come before. We have a plenty of choices to explore the city. We can go straight to the tourist bureau to ask for information and pick up a map or do just walk into the crowd and let us instincts take over. However, A Tour is always a good choice because we will be unlikely to get lost and waste precious time and money and miss out on important things to do or see.

Touring in exploratory testing is the same concept. Tour testing gives us the opportunity to study our product from a different frame. During the tour, we again more knowledge about the product with different perspectives and obtain a richer pool of ideas, data, and questions that contribute to an overall better understanding of the product.

There are some implications for tour testing:

* Requirement Tour: Find all the information in the software that tells the user what the product or certain feature does. Does it explain it adequately? Do results reflect the claims made?
* Complexity Tour: Look for the most complex features and data, in other words, all places where the most inextricable bugs could lurk
* Continuous use tour: Leave the system on for a prolonged period of time with multiple screens and files open. Observe what happens as disk and memory usage increase
* Documentation tour: Tour the help section of your product and follow some instructions to see if they produce the results desired
* Feature/ Menu tour: Try as many of the controls and features available on the application as possible
* Inter-operability tour: Check if the system interacts as it should with third-party apps and whether data is shared and updated as it should
* Scenario tour: Create a scenario (user story) that mimics the real-life interaction of a user with the system and play it out
* Variability tour: Look for all the elements that can be changed or customized in the system and test different combinations of settings
* Rained-Out Tour: Start and stop tasks, hit cancel, etc.
* Saboteur Tour: Perform software to carry out tasks like: Access a resource, Open a file, Perform a Task; understand how the resource the software uses: memory consumption, Files Access, Network Types/ Transfer,.. then prevent the software from doing its task like: Take away memory, delete/ rename files, disconnect the network
* Supermodel Tour: Only skin deep; ignore all functions and stay focus on UI/UX. Answer questions like Does it render properly? How does it look? Are all elements consistent across the application?...

Here is a practice of combination of testing tours used by Nhu Le and Phuong Le who are now the exploratory testers at MeU Solutions, it is described in a session as following

“*We started going through key features (feature tours) to get familiar with controls/ pages and menu items from individual feature. We also tried to understand how these features connected in somehow. During this tour, we simply asked questions like “What is this and what it used for?”; “how we know if this feature (control, menu item) is working?”; “why is it working in this way?”… We also seek for interactions, calculations, transformations, error handling,.. and take them all noted by a mind-map*

*Next, at each page, we looked for things changeable, and make them changed with different variables or data to see what was happening and get a feel how the things work, and possible values might be. These changes might cause an alignment for our model which had been built in our first feature tour . Then, we would obtain the product knowledge accurately (this is using Variability Tour)*

*With complex pages or complex features such as: pages with many controls and variables, features using complex algorithms as manipulating data to get insights into them*”

* <https://www.tricentis.com/softwaretesting/types-exploratory-testing-tours>

<https://techblog.appnexus.com/exploratory-testing-et-what-when-why-and-how-a29a879cb20e>