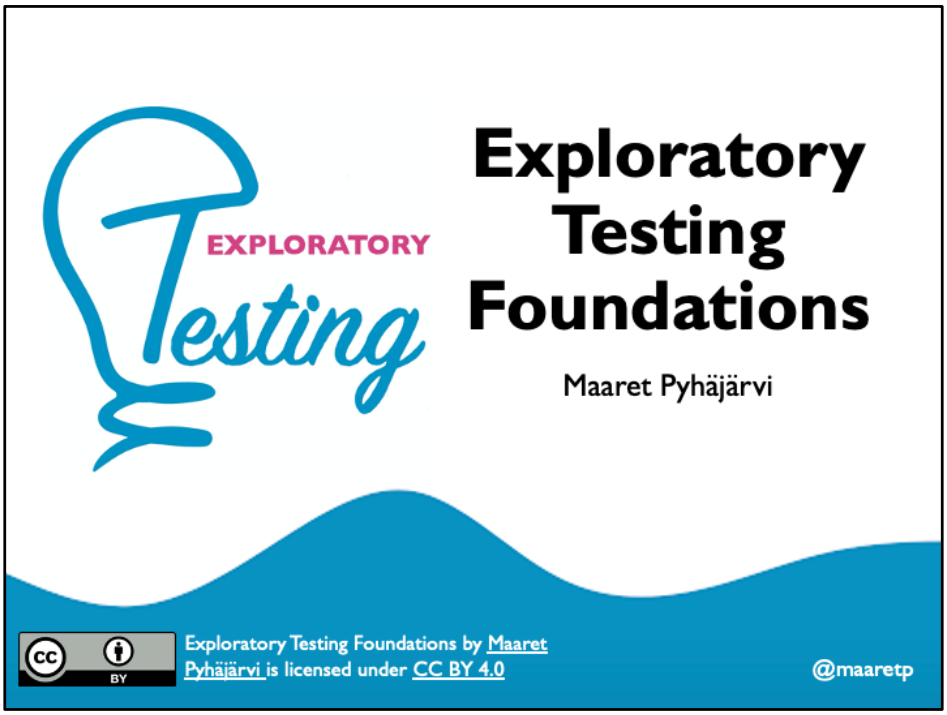


Welcome to Exploratory Testing Foundations –course. This course intertwines a simple application to test with basic theory of how to do exploratory testing to give you a foundation to build on.

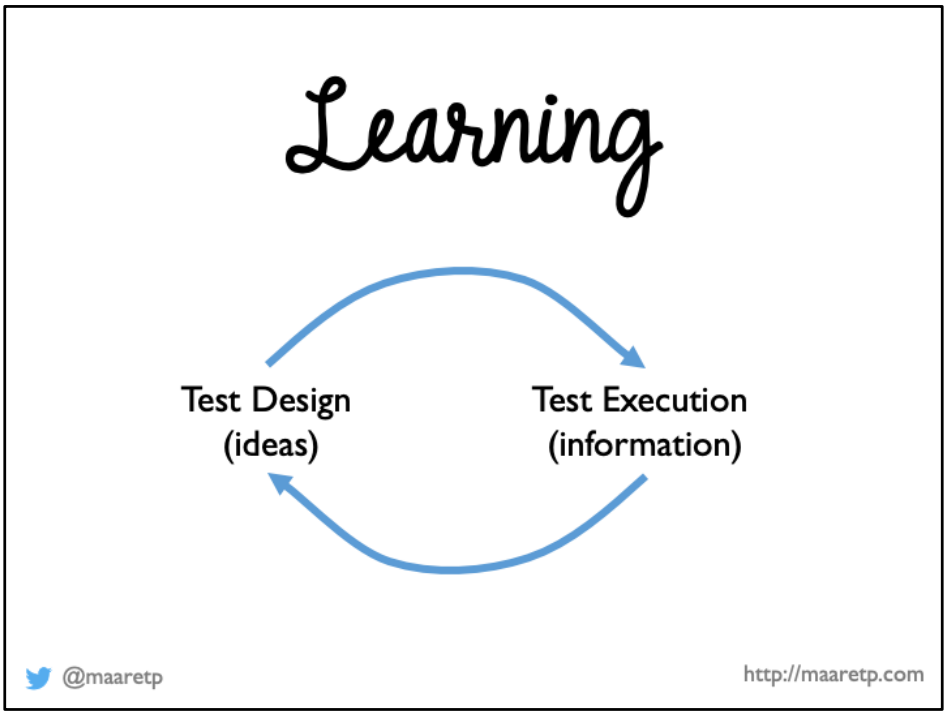
Exploratory testing is an approach to testing that centers learning. Test design and test execution form an inseparable pair where the application and feature we are testing is our external imagination. It takes domain knowledge, requirements and specifications, and testing knowledge as input and produces information and a better tester as an output. It usually also encourages us to at least consider documentation and test automation as a form of documentation.

I think of this course as an antidote to the idea that test cases tell you how to test a feature and that is where a new tester would start. That type of test cases are only a small subset. You are expected to find defects, where the system does not work as we specified but not stop there. Finding change requests, things that would make things better for users is included. And instead of using most of your work time on documentation, I'm inviting you to consider lighter and executable formats of documenting.

This is what we fit into one day with one application. Theory and application go hand in hand. When taught in classroom, we will also reflect course experiences to work

experiences and share war stories of testing in projects where applicable.

Exploratory Testing Foundations by Maaret Pyhäjärvi is licensed under CC BY 4.0. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/



Exploratory Testing centers learning. The application we test is our external imagination. While we test, we learn about the application and about ourselves. We optimize the value of our work continuously. Instead of following a plan we created at a time we knew the least, we create plans, learn, adjust, even completely revise as we learn. Our ideas of the plan are best when we know the most, at the end of our testing.

To emphasize learning, we emphasize agency – the responsibility of the person doing testing to do the work to their best ability, and to grow with the testing they do.

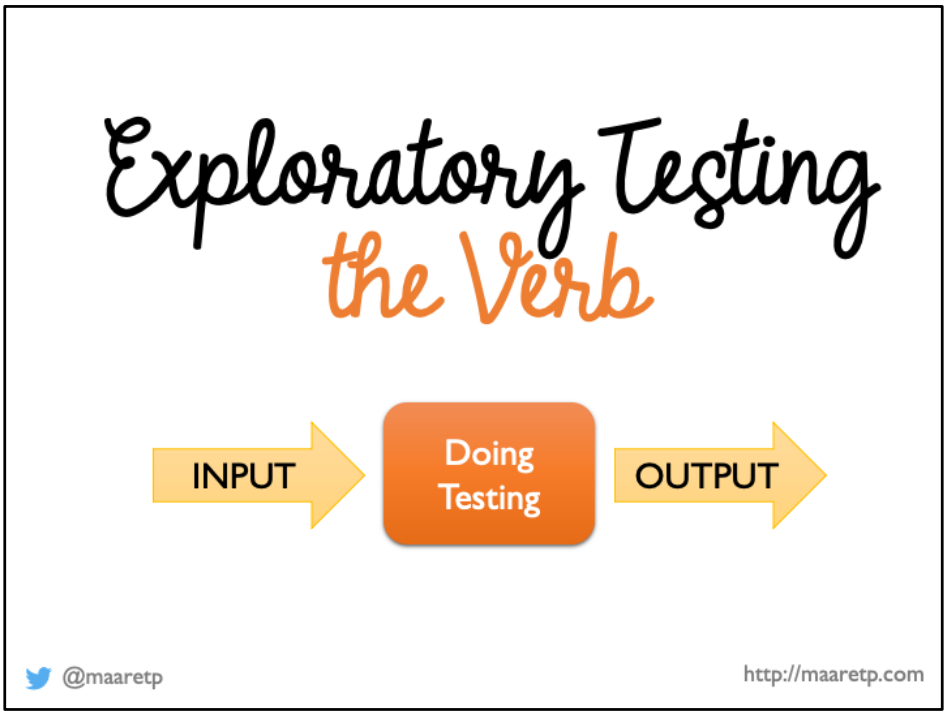
We remember we learn

- by researching the domain – the business, the legal, the financial
- by passing on information from those who worked on the problem before us - the stakeholders, the requirements and specifications
- by using the application and thinking while using it
- by critically evaluating the application as we use it
- by focusing our attention to both how it could work and how it could fail
- by reflecting new information against what we know from the past
- by experimenting with approaches outside our usual repertoire

The work we do is done in slots of time: a minute at a time, an hour at a time, a day at a

time. Every unit of time, as we learn, makes us better at optimizing our value.

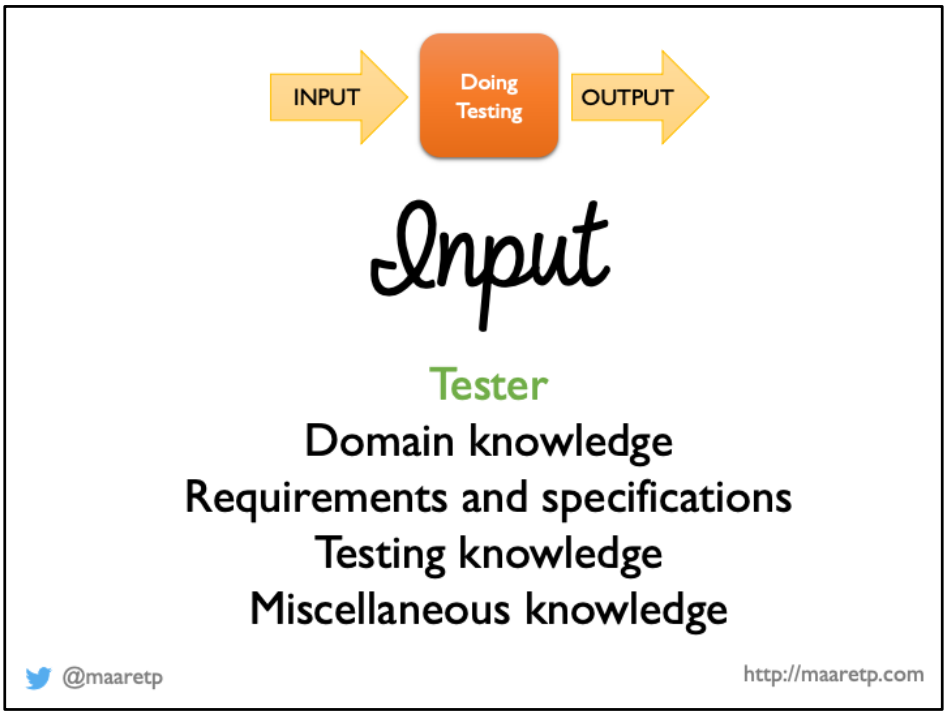
We go as fast as we need, or as slow as we need. Just like driving a car requires you to take yourself and your surroundings into account to choose the right speed for the situation at hand, same applies with exploratory testing. You drive, your speed and your route – with the destination in mind.



To set our minds for the road, let's still talk about the process of exploratory testing.

We are given something new to test. It may be a new application like on this course. It may be a change in the application you've worked years on already. It may be a new feature that is new, in an application you are already familiar with. Your task in testing it is to provide information on when it works and when it doesn't, in the perspective of stakeholders.

This is a process of information discovery. We already know something, and we use that. But we are asked to learn more and share our learning with the rest of the application team.

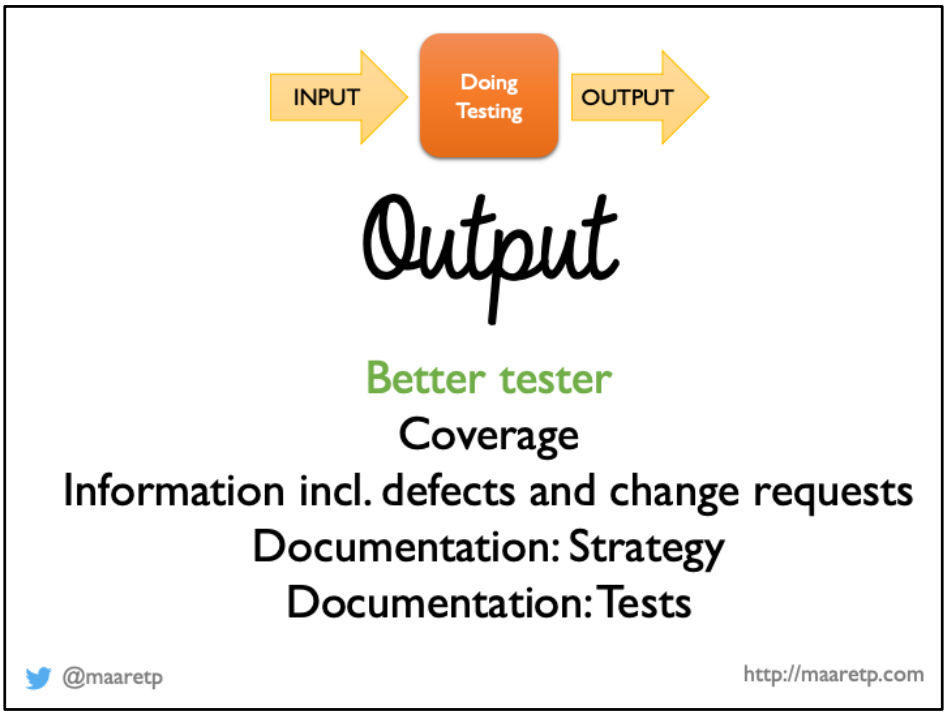


The most important thing going into the process of exploratory testing is whoever is doing testing. It could be a career tester. It could be programmer. It could be an analyst. Job role aside, we call someone who performs testing a tester.

No matter what the features the tester goes in with, going with the idea of learning while doing, we come out different. Since we optimize our value, we optimize our learning too: we want to be always both **learning** and **contributing**.

What we can't take in as we start, we can acquire as we do testing. We can ask around. We can research. We can read existing documentation and apply it to the application under test. We can constrain ourselves with a test technique. We can make notes and create versions of documentation we intend to leave behind.

Exploratory testing – doing testing from whatever the input - emphasizes learning while contributing. And while it can be something you do solo, on your own, it is also something you can do with a pair or a group.

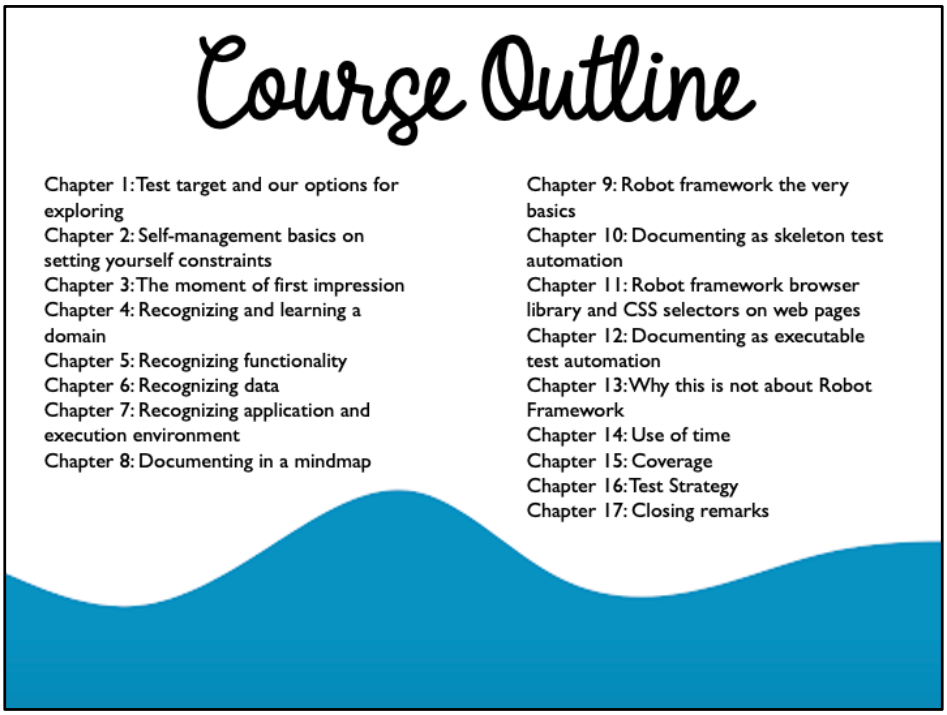


What comes out of the process is a better tester but also different kinds of deliverables we consider relevant when optimizing value of testing for *the specific testing at hand*. Sometimes we only report new information. We leave coverage as implicit, but our information is only as good as our ability to cover multiple perspectives. While we can come out with only information, we often want to put our learning into a good use for our own and our coworkers' future and leave behind documentation, even executable documentation.

Let's still say this out loud: test automation belongs in exploratory testing. You can't explore with great coverage without automation. You can't automate without exploring in a relevant way. When automation fails, it calls you to explore. With that said, we are optimizing the value of our testing with exploratory testing: value today, value in the future. Automation is a constraint that directs our attention.

Exploratory testing can include activities *you* personally are not able to do. Sharing work with others, collaborating, is advised. Also, I advice growing your personal abilities, and remind you that contributing to test automation efforts isn't the most complex of our programming tasks and believe everyone can learn it. I am less sure if everyone can learn to think in ways that ensure multidimensional coverage and I'm hopeful with that too.

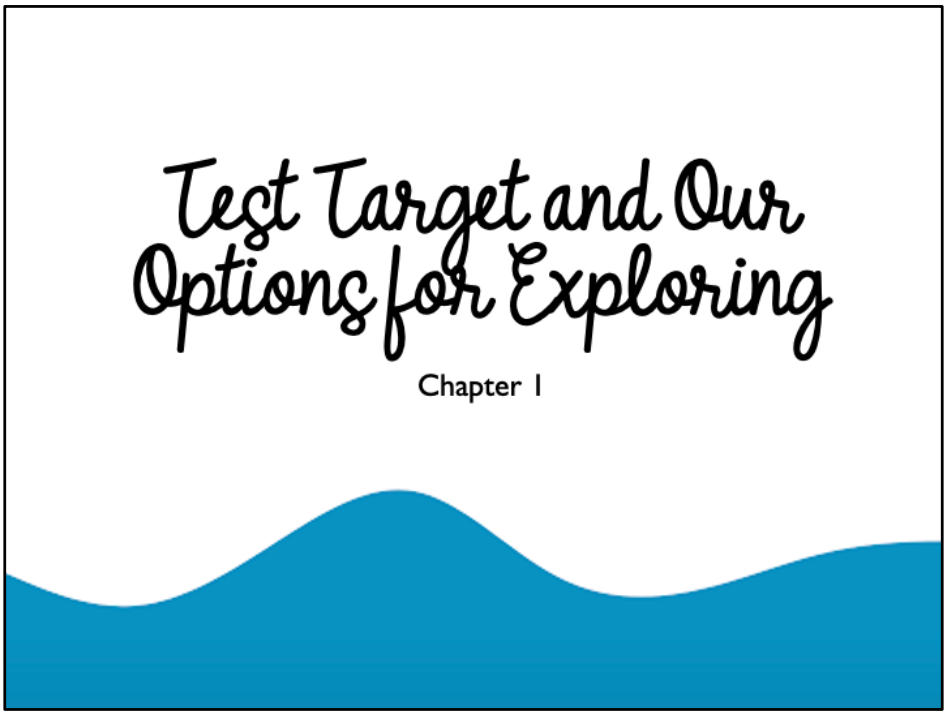
Successful output is effective: we find problems our organization expects us to find. The path to the result of finding relevant issues is through understanding coverage.



This course is first in the series of Contemporary Exploratory Testing courses where teaching on the course is framed around a particular test target. For foundations, our application is very simple, and it was chosen based on the idea that is *not supposed to be full of bugs*. While finding lots of bugs is fun and does wonders to a tester's self-esteem, it gives a false impression of what it is we do when we do exploratory testing.

The course is split into 17 chapters, where each chapter will have a supporting video, and the final chapters will describe the best testing we have done on the application. It isn't a recipe for all applications, but an example for you to understand what coverage for this application may mean, and what parts of the work you could do make sense when you are truly optimizing value.

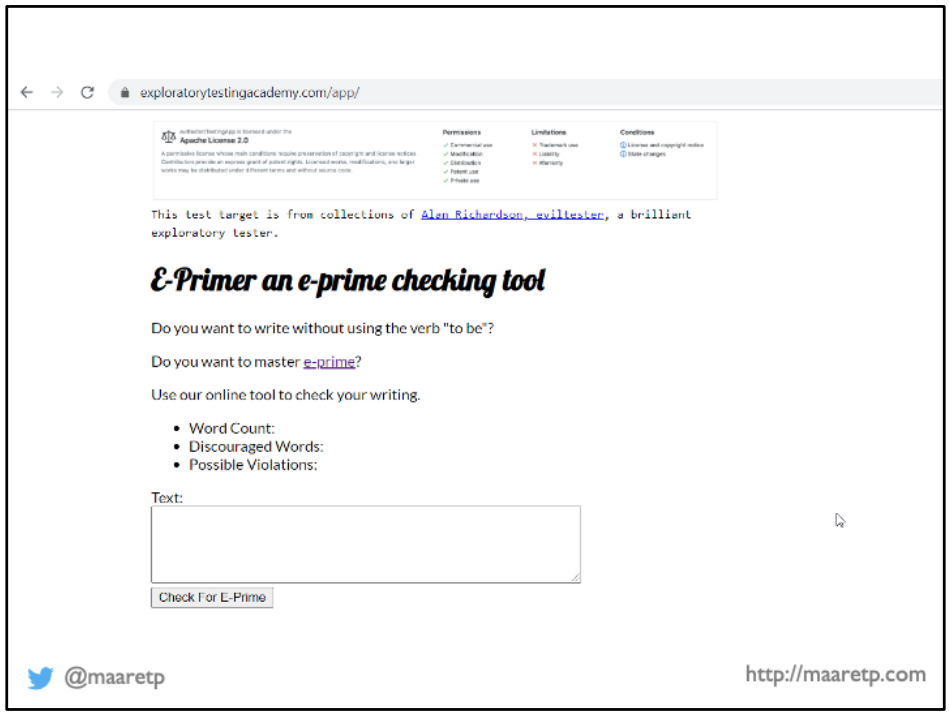
From the outline, you can see that we will be using a test automation framework (Robot Framework) on this course. Choice of tool is irrelevant but using one to give you concrete examples helps teach this content. Robot Framework allows for very natural language like programming in its own custom-made language and can be useful for new people to get into test automation programming, even if I believe it soon becomes a limiting factor you may want to learn away from.



In this first chapter, we don't yet get to use the application, but we get to see the application. Sometimes a good way to learn about how learning changes things is to try to think what you would do before you had more information.

With exploratory testing, you need to appreciate **learning**. Learning shows up by your ideas changing, and ideas changing change your actions. You should notice when your ideas change. It is expected, welcomed, and makes you better. You can't have the best answers available at the time you know the least.

With an application you have never seen before, it is clear you now know the least. It is harder to appreciate how that is true on your day-to-day job, with the same application with new changes coming in. Yet it is the same: at start you may know a lot, but you still know the least about that specific change and its impacts on the application compared to what you can know given proper time to explore it.



The application we test on this Exploratory Testing Foundations course is called E-Primer. It's a little application for people who want to check their English writing to master e-prime – a way of writing English avoiding the "be" –verb in all its formats.

I chose this application because I was under impression it is not target-rich application for testers. That is, it is not so full of bugs that you should consider it ridiculous.
Having tested it, I know it has its share of issues.

We can figure out what the application is and what it does by using it. Also, name of the application gives us a hint and allows us to research e-prime further should we want to.

By the time we get to chapter 3: The Moment of First Impression, your first impression on just seeing the application is already gone. But the first impression of using it is still ahead of you.

Before you move forward, stop to think. What would you do first, and what soon after you get started? If you make an inventory of ideas you have, what do you list? Try doing that.

We, people testing applications, come to the targets of testing with our biases. Our internal dialogue of being awesome just as much as our internal dialogue of being bad have impact on our ability to look at what we can do objectively, which is why I encourage writing things down to support your own learning about yourself.

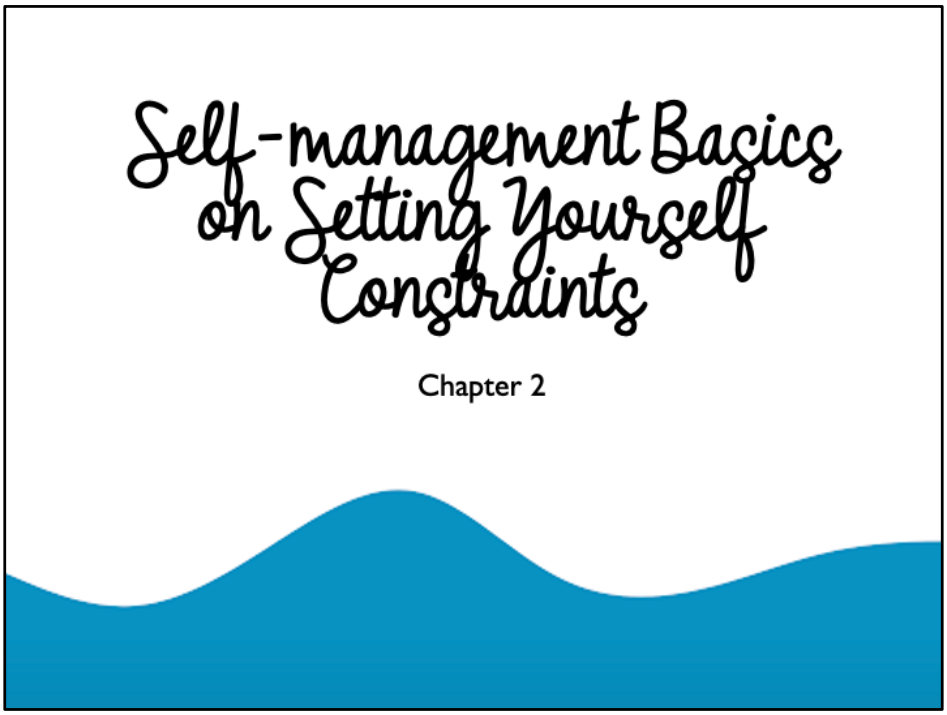
This supports your learning, and I would not ask you to do this with every application you ever test. But it is an option you can start with, an option I enforce here for learning purposes even if it didn't help with optimizing value of testing the application at hand.

Now that you have created an outline of what you would do without doing anything but digging into your past lessons, it's time to discuss your options for getting started with a new application.

Time you spend away from application to learn about it is time when you don't know about what the application does.

Time you spend wandering without a purpose with an application could be a way of learning about the purpose, but there may also be more effective ways to learn about it.

Balancing your options and creating new options as you go is at the heart of exploratory testing.

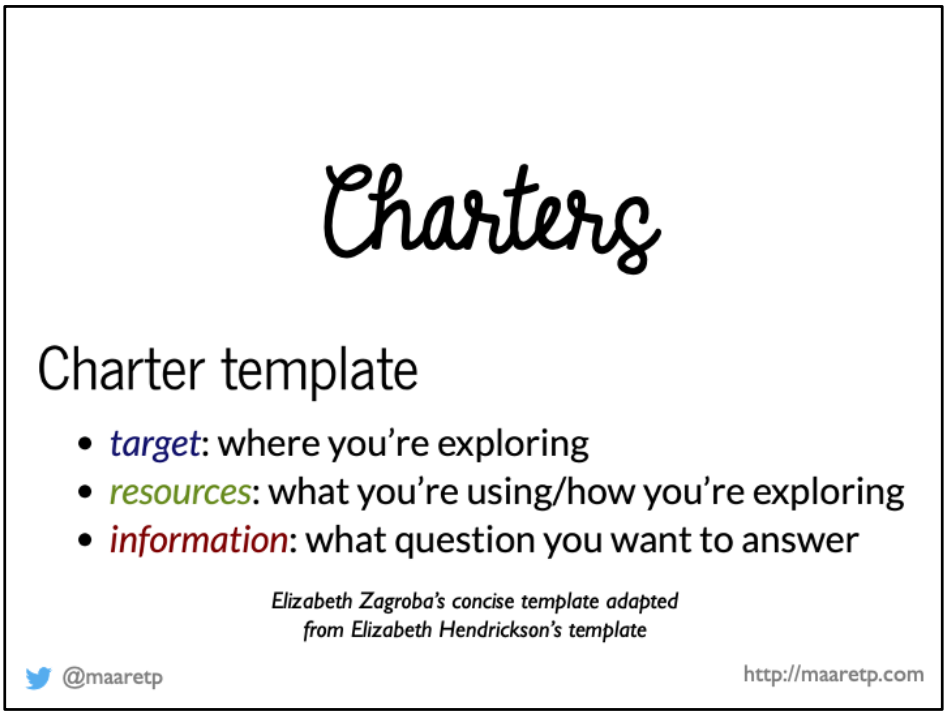


You have option for exploring, and your task in exploratory testing is adjust your constraints on a cycle that enables you personally to keep up and do the best possible testing you can. Exploratory testing centers the tester, so you don't have someone from the outside telling you what detail to verify, the control is with you.

Some people tell me that the freedom frustrates them and makes it hard for them to start. They don't have to have this freedom; they are free to set themselves into a box that enables them. They are also free to let themselves out of the box they created, when they discover that to be useful.

In this chapter, we introduce three concepts:

- Charters
- Constraints
- Multi-dimensional thinking for intent and learning



Charters are a key concept in how the community talks about framing our exploratory testing. You can think of a charter as a box that helps you focus and generate ideas, but also assess when you think you are done with a particular idea.

I like to think of charters as free form test cases.

A charter could be a chapter from your design specification you want to understand empirically with the application.

A charter could be reminding you of a non-functional perspective, like accessibility – can people with disabilities, permanent, temporary or situational, use the application, being an important group of stakeholders.

A charter could be a very traditional step-by-step test case with your promise of stretching every single step to both it's intended path but paths it inspires.

Charter is a structure for thinking like this, not passing work along like this. Some people use charters to share work in team and my advice is to not do that unless you co-created the charters in the first place. As soon as you remove tester designing, executing and learning intertwined, and replace it with a tester designing and another executing, you will

lose a core feature of exploratory testing.

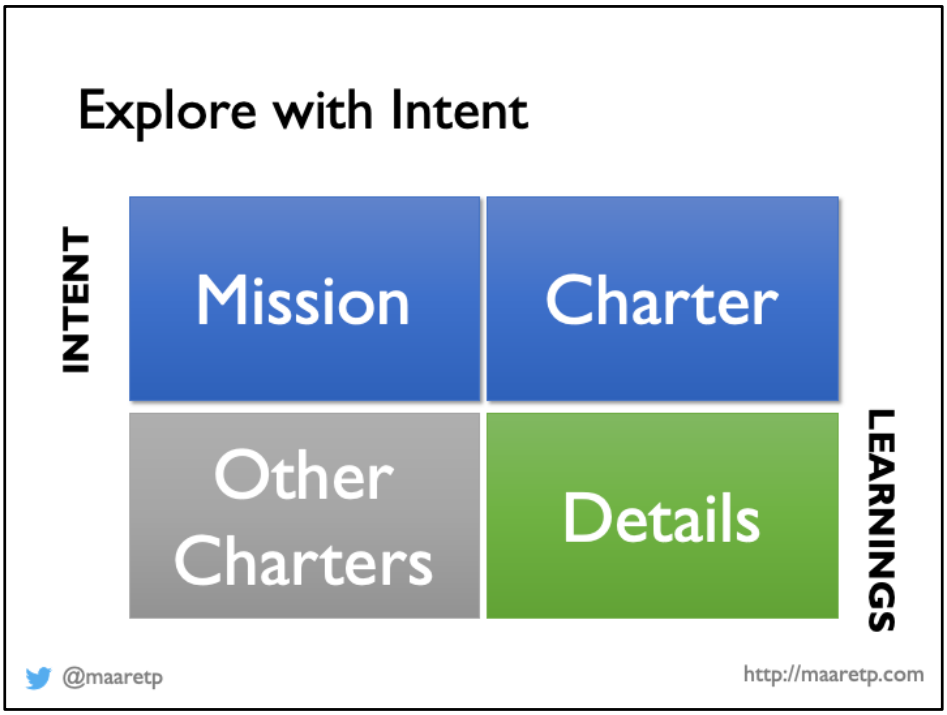
Elizabeth Hendrickson introduced a charter template in her book Explore It, and Elizabeth Zagroba introduced an adaptation of it in one of her presentations. I like the concise template a lot but encourage you to think in terms of charters being anything that can box your testing and help you maintain focus, rather than follow a format.

I prefer framing exploratory testing with constraints, which is one of the options to use in *resources* part of the charter template we looked at. The way I look at it, we have limited bandwidth to doing many things at once, and for us to frame the work we do in an effective way, we need to deliberately exclude perspectives.

Like, I could say I am deliberately not writing notes at all to get through as much of different scenarios of using the application I can think of. Or, I could say I will go through a document describing correct behavior in detail, ensuring not to miss important claims.

If I try to do everything at once, I can't get any of it done. So, I get to choose my constraint with my primary heuristic to never be bored.

Testing is a lot of fun. Finding out information that others don't know is investigative work, and servicing many stakeholders is an intellectual challenge. No matter what changed, there is something new we now need to figure out, in a way that optimizes the value of our work. Doing the exact same things isn't optimizing value of our work, so we start with the idea of always varying things to not be bored. If you find yourself to be bored, you need a new constraint that challenges you.



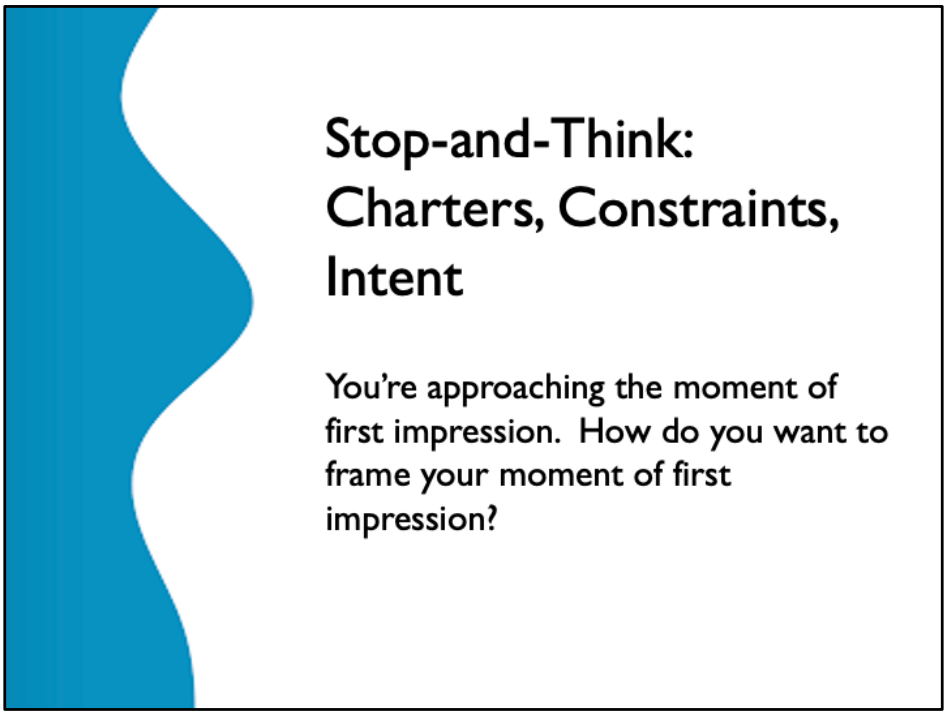
Exploring is wandering with a purpose – without purpose, we are lost. A high-level purpose of spending time with the application or finding information is a little too generic. Specific purpose, *intent,* looks at what is the next step, next timebox and next theme, balancing serendipity (lucky accident) and coverage (making accidents likely).

When we practice being intentional, learning to structure our thinking that is multi-dimensional can be helpful. When you imagine and fill the next slot of testing in your schedule be it next hour or next day, I find this matrix has helped me keep track of my intent and learning.

Think of this as an empty A4 paper for the next piece of testing you are about to perform. **Mission** includes a statement of why you exist in the organization, what is expected of you – there might be a role-based constraint applied to you such as being a test automation specialist, but that does not stop you from doing other things, merely reminds you that if there is a priority call, yours might go this direction. Like a sandbox you are responsible for, but with sides so low you can easily cross them when you are excited, this corner reminds you of where you anchor your purpose in larger scale. **Charter** would talk about something on shorter term, like a personal promise: "today I will test with 100 different inputs on one browser", implicitly saying you will do that rather than test for example 10 inputs on 3 different browsers. It's your intent you are framing, so it is your right to change the framing

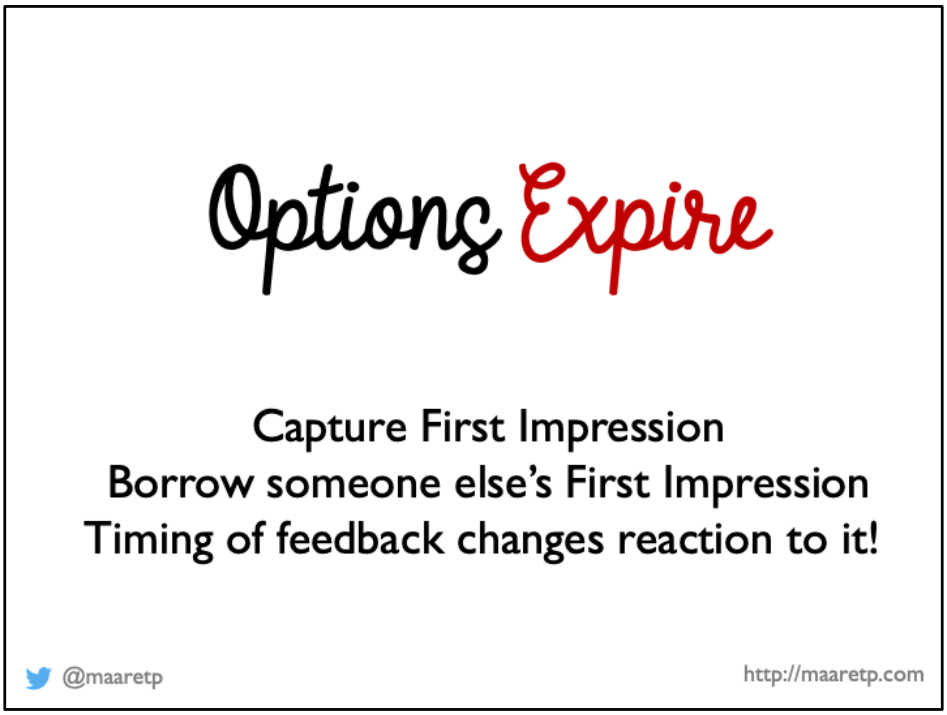
any time. **Details** makes space for things you note while you test on level of details. Typically, we note things that are bugs, things that are questions, and things that are ideas for documenting for future in whatever test material I leave behind. **Other charters** is your placeholder for things you'd like to do but won't do now even they popped up in your head under influence of the application as your external imagination.

When you find a bug, you can choose to write it down quickly and continue. Or you can choose to isolate it properly and log it. Your choice.
When you find a question, you can choose to ask it right away. Or you can choose to collect them for later, seeing if other things you learn while testing will provide you an answer.
When you find something to document, you can write it down in the best possible format immediately. Or you can write a note appreciating it as something worth documenting and take time later.
You can also completely skip a bug, a question or the documentation. The more thoroughly you process them, the more time they take from what your intent for this testing was.



You are approaching the moment of first impression. You usually want to see it work before you see it fail, or you don't understand and appreciate the way it fails to communicate the fail properly. What is the first thing you will do? Would you frame your start as a charter, a constraint or as an intent? Or maybe you have your own style of framing your testing that we did not talk about here?

We already saw the application, even if we did not yet use it. One more thing to note before heading in for the test. You are only new with an application once. Even if it is time when you know the least, let yourself listen to how you feel about the application. Your joy could be user's joy. Your confusion could be user's confusion. Your mistakes are most likely some user's mistakes.



With making choices of what we do first and what we do after, it is good to be aware that while we have an endless selection of options, some options expire.

You can only have a 1st experience without having read documentation or given a personal demo before those have happened for you. Even if you can't have the first impression personally working closely with the feature, you can always work to borrow someone else's experience through pairing or watching them use the application.

While many of our options don't expire and we can do them in which ever order, it takes exceeding amount of energy to remain curious about the new information when you have already tried many things. I find that we often give up and stop testing too soon! In the words of the famous Albert Einstein: "It's not that I'm so smart, I just stay with the problems longer." Given time of use, software has the habit of revealing issues it has always had but we either could not see or appreciate earlier.

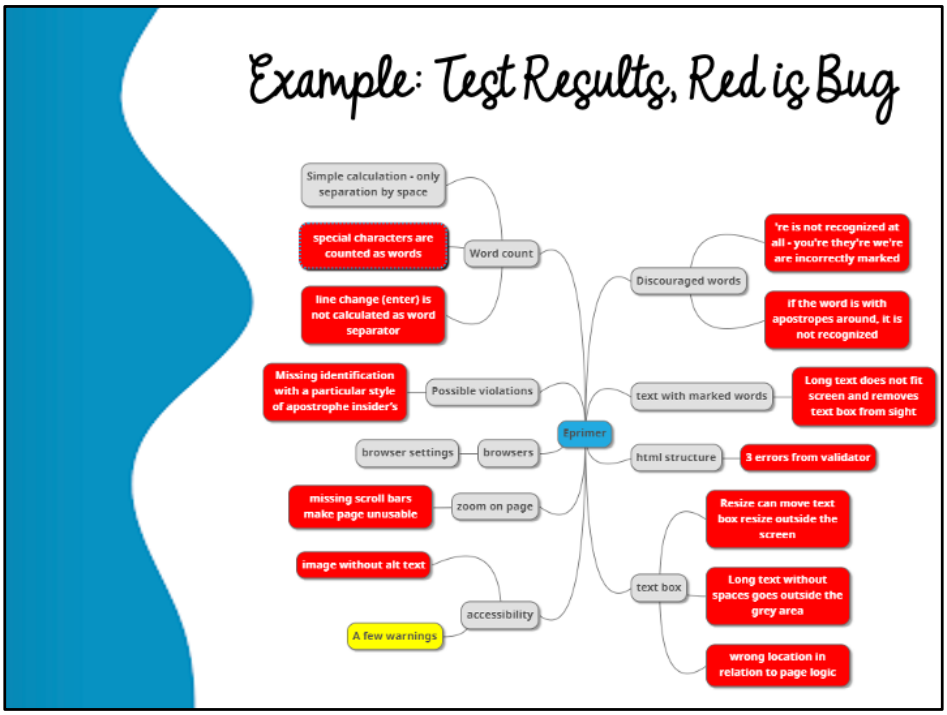
When working close to a deadline, options expire also on what feedback is welcome. Days into a major release, issues considered major months before can be prioritized down. Timing of feedback matters.

You usually stop testing before you have exhausted all your options. Knowing your options

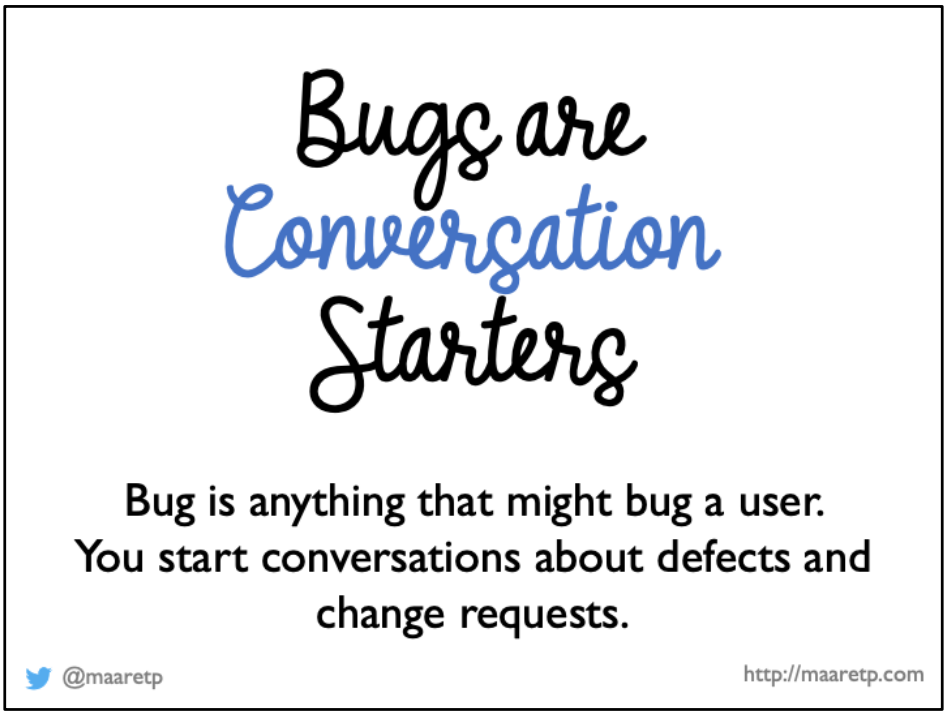
helps you be intentional about it.

With all this said, it is our time to test. The application is available at https://www.exploratorytestingacademy.com/app/ and eviltester's original at https://eviltester.github.io/TestingApp/apps/eprimer/eprimer.html.

Pay attention to where you start from, how you learn, and how you decide on your next steps. Each of the following chapters gives you one constraint that is applicable to this application at hand and explains where the constraint leads you.



Your first impression and learning of the application is most likely not taking you this far. This mindmap is from an hour of pair testing with a tester who found none of these problems alone but needed training on how to look at an application to identify what might go wrong with it. The mindmap is by no means exhaustive list of things that are off with the application. We'll talk about bugs in more detail when we get to the part about coverage – because most relevant coverage is bug coverage except that we can only assess in hindsight.



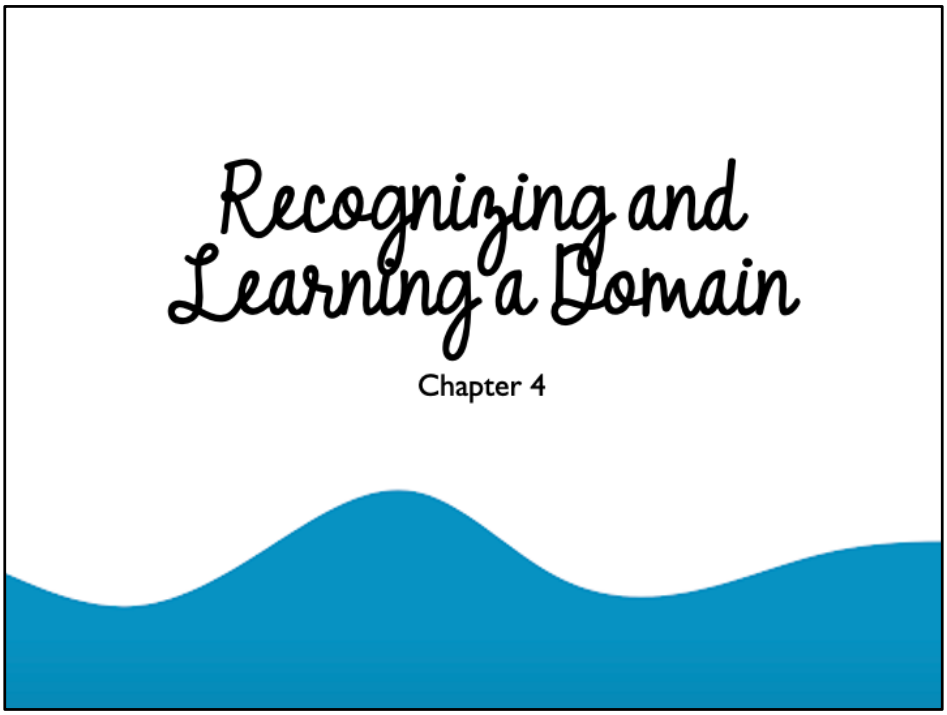
We would hope first impressions are not about bugs, defects and change requests for users of our applications. Tester's sometimes get to bugs on the very first thing they do with an application, drawing from past experiences to select a way of using the application that reveals a bug. Tester's fast forward a user's year into the hours they use testing – or better yet, multiple users' years.

We like to think of bugs as conversation starters. Sometimes we need to avoid the word defect and reserve that only for use where it is clear that the behavior of the application is against something we explicitly agreed. Defects come with the burden of guarantee, whereas change requests have a different tone to them. In many teams, we have called bugs "undone work" to remove the judgement from the conversation – it is simply undone work we propose we still might want to do.

Bugs are things that might annoy a user, and unless someone starts a conversation on them, they conversation starts only at a time a user starts it. These conversations turn things we didn't know into something we can be aware of. And for many, it enables us to design changes that remove the bugs.

You can also start conversations on the good things you see, the absence of bugs that surprises you or just the things you find that make our users with our application more

awesome. Having that empirical touch on what we have built gives you a perspective of use that serves as a conversation starter. Identify something good, find ways of getting more of that kind of good.



Let's move to discussing domain, requirements and specifications. The information about what the application is and does comes to us in many forms.

Domain is the problem-solution space of the application. The application exists for a purpose, and that purpose is the domain.

Sometimes we know the domain by just our life experience – like most editors. We know what a text editor does. While there are specific functions to a text editor, we can figure those out since we too have edited text before.

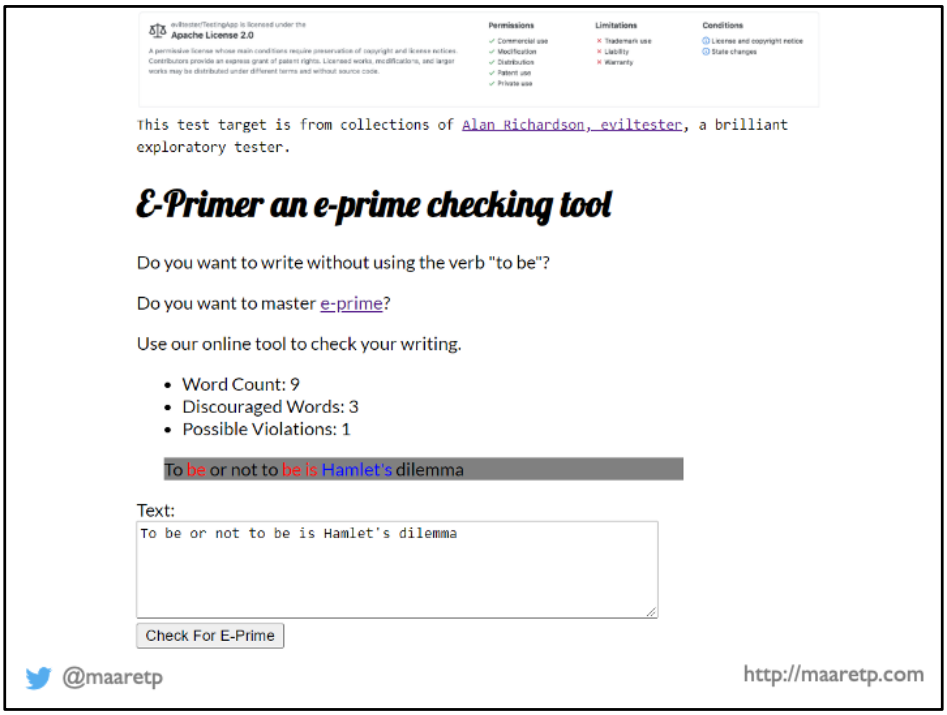
Sometimes we know very little of the domain and need to learn it as we test it. Learning a domain effectively and becoming a domain expert over time is what we would expect from someone testing applications in a particular domain for a longer period. Knowing the domain or learning about it until we know is what separates us from assuming something could be right to understanding if it is right.

To learn a domain is to acquire information about the problem-solution space of the application, and usual expectations of it.

You have three main routes to it:

- **Conference** is about asking around. Talk to anyone you need to.
- **Reference** is about getting an authoritative document. It may be given to you directly, or you may use general search to find it.
- **Inference** is about applying other knowledge you have access to with this domain, expecting similarities or differences.

You may have a requirements specification. You may have a functional specification. You may have a user interface specification. You may have an architecture specification. No matter what you have, it is not all you will need. And if you have none or some of these, or some others, we advice to think that documentation is useful but also it reflects the things we already think we know. Exploratory testing begins with what we know and seeks to learn what we don't know yet.



With E-Primer, we don't have a specification. We have a Wikipedia description of the domain that gives us a decent perspective into what the application is designed for, without knowing any of its intentional limitations.

We can search online for any information about e-prime we consider useful and educate ourselves on it.

With specification, we can find phrases to test with that showcase the application's functionality. This good demo phrase is a result of testing, not the first idea to use the application even based on its specification. We can best get good demo examples by asking the developers on how they discover the functionality.

With this specific application, we can find the source code to view it in the browser.

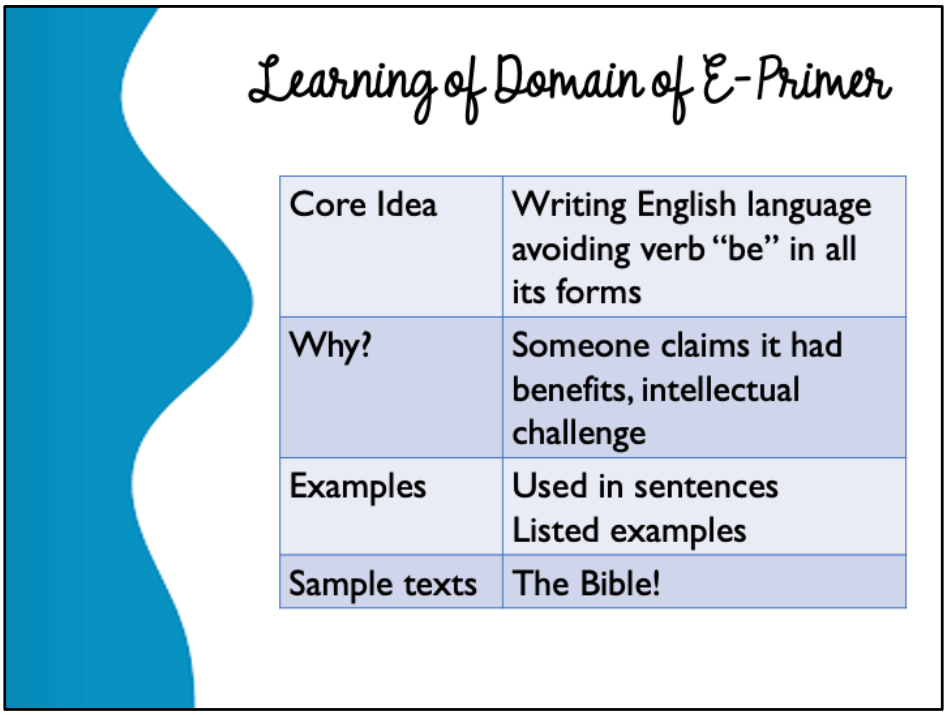
Or we can find it in GitHub where it is hosted. https://github.com/exploratory-testing-academy/ETF/blob/master/app/eprime.js

Looking at the code gives us a direct chance of comparing what it can do to what we would expect it to do with regards to the Wikipedia page. We could, just by reviewing the list of words that get marked discouraged identify that the words “you're, we're, they’re” that should be marked won’t be, as they are missing from the list.

Programs do what they are told. You don’t have to know how to read all of code to read enough code to make sense on existing logic.

We recommend all testers read pull requests / commits to understand scope of changes they are testing.

It is time to test. Try out what how testing flows when focusing on the domain knowledge. What do you think of coverage of what you have tested so far? Eventually, we care about you not missing relevant information others don't yet have and thus can't specifically ask of you!



The combination of reading and using the application means a significant portion of your time went into making sense on the Wikipedia text and finding other relevant references. You most likely learned something.

We expect you are now more comfortable with the application and have an idea of what it does.

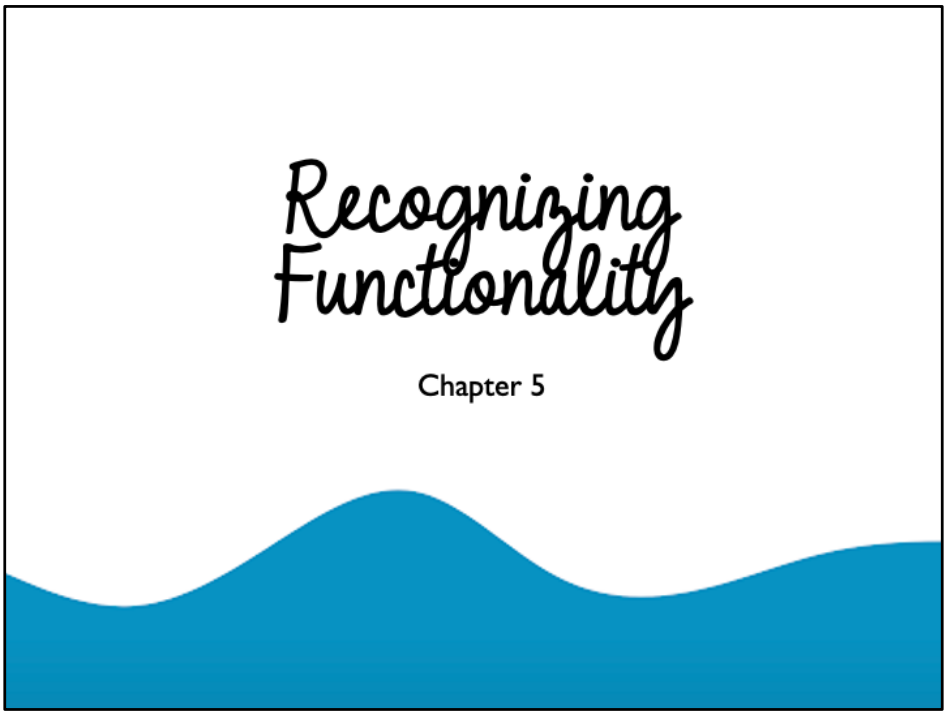
You may still find the "Possible Violations" something you can't understand or figure out. The written references on it are not particularly helpful.

Most commonly people come to understand "Possible Violations" as a side effect. We commonly see three routes to finding it.

1. Using the demo phrase this course has – given the demo phrase, people miss out how hard it is to figure out when it should not be
2. Reading the code – figuring out from the code that "'s" is the search criteria for counting things as possible violations, and then understanding that it looks like possessive and "be"-verb appear so often in hard to distinguish formats that this appears a likely design choice signaling need of user intervention over programmatic algorithm doing all the work
3. Using large data samples – browsing through large data samples, like copy-pasting the

whole Wikipedia page text at once in the application and browsing for blue

You learn domain by asking questions and paying attention to answers. Not all your questions have an answer, but they start conversations you get to consider how far you take them. What helps you optimize the value of your work when testing?

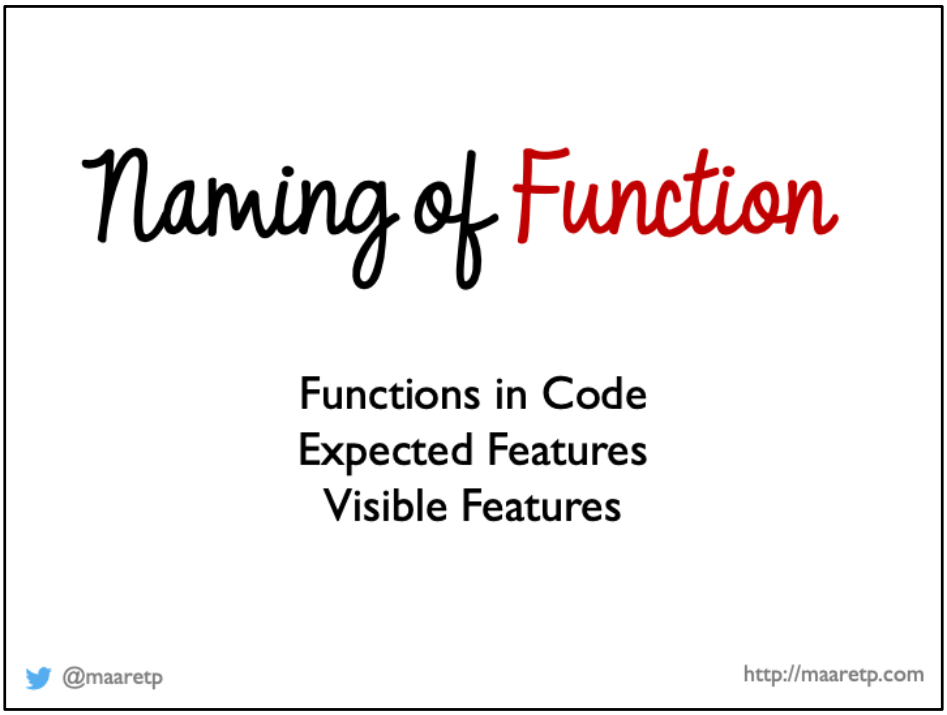


Now that we have an idea of the domain, let's look at the functionality.

Different applications are built on a different technology. As someone testing an application, knowing about the technology is something we would expect. Not expert level knowing, but at least a basic level curiosity turning into expert level knowing over time, question after question.

A lot of testers given a picture of the application start suggesting SQL into the text box, which makes little sense now that we know the application under test is a javascript application that runs in your browser after you have downloaded it. The application keeps working even if you disconnect from network as long as you don't try to refresh from the download location.

Let's discuss functionality as constraint a little more.



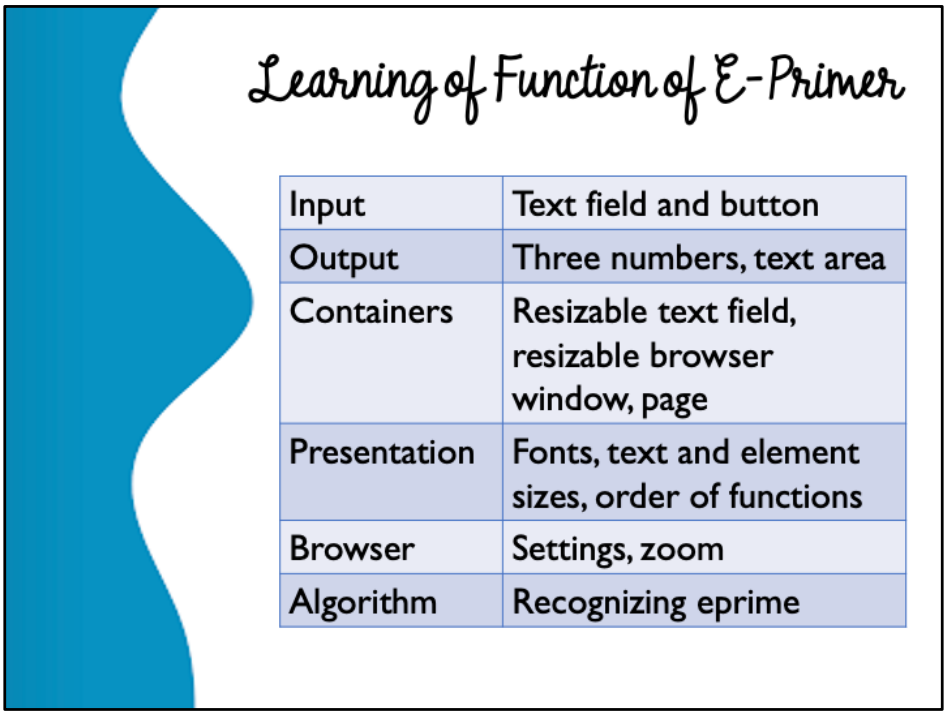
For functionality – features – we find a good approach to be seeing function in different scale and naming it.

The code has functions the programmer has given names to. It is structured as functions the programmer calls to get their intended results overall. We could use a unit testing framework to explore the functions of the code, and for the scale we can, we probably should.

The application has functions we would probably name ourselves. Some of those functions come from what the programmer explicitly introduces, some come from the fact that it runs in a browser.

We can name what we see. We can name what we expect to see. And we can compare those.

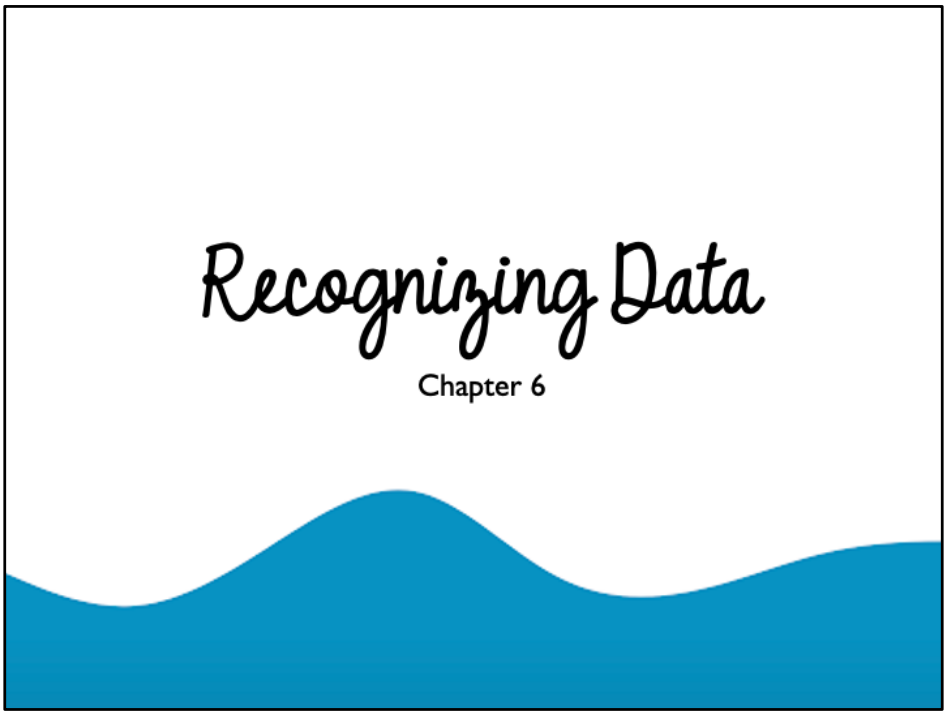
Given the constraint of function, what would you test?



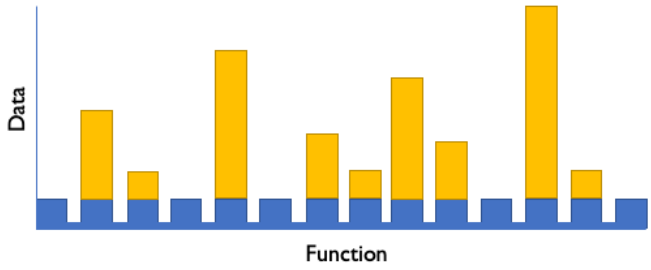
Usually when we use the constraint of function after using the constraints of domain, we find there is either an overlap or challenges in naming things we did not already see. Many times, we find we need to point at a function directly and name it to make it visible.

With this application, there are a few functions that are not obvious.

- Text area for output text has a size limit with the grey background
- Text box is resizable, and so is the page and browser window
- Anything running in browser has a connection to browser settings
- The eprime recognition algorithm is core to the problem and feels to be on a weaker side
- Between our version of the application and eviltester's version of the application, we have lost scroll bars – a function really relevant when dealing with larger chunks of text


Recognizing Data
Chapter 6

Data or Variables



@maaretp

http://maaretp.com

Versatile Data

Lifecycle of Data: Create, Read, Update, Delete

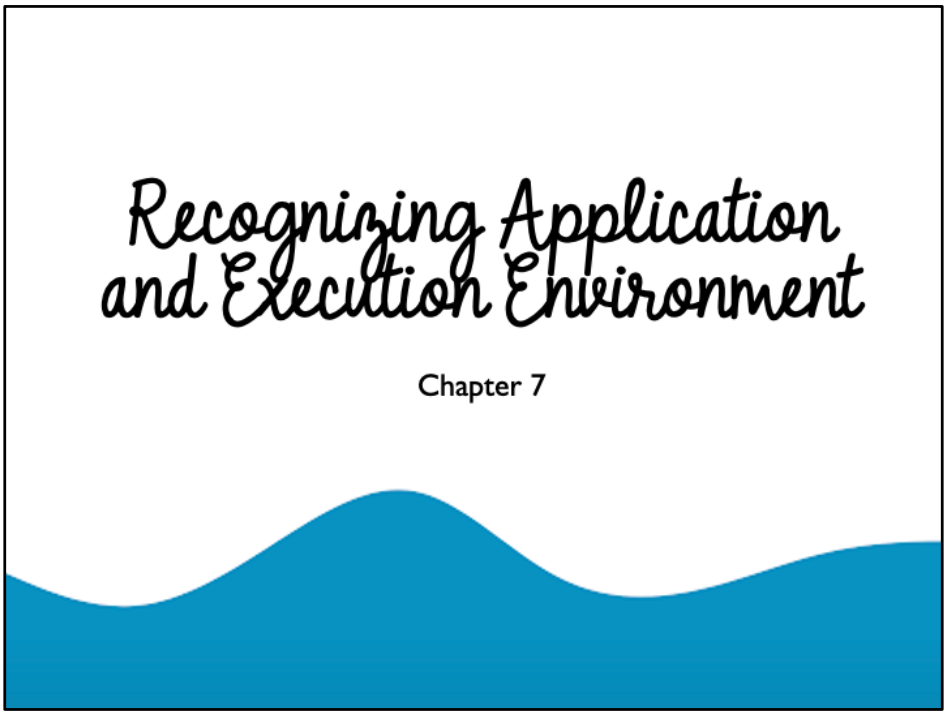
Known problematic inputs: GitHub Naughty Strings

https://github.com/minimaxir/big-list-of-naughty-strings/blob/master/blns.txt

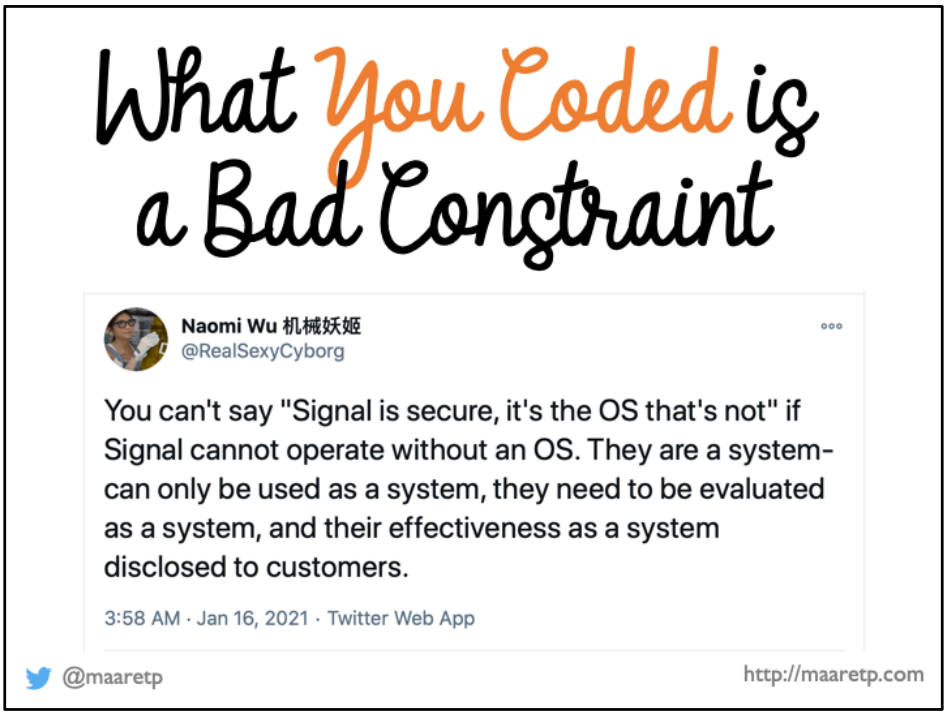
Given the constraint of data, now it is time to find the things that could be different. You find data you can vary on the easy places like input values to a text field, but equally data is there on getting to output values of the application. What can you vary to discover problems the application holds?

Learning of Data of E-Primer

Word delimiter	Space, wordcount breaks with characters and line change
Types of apostrophes	Typesetter / typewriter
Long text	Copied / tool generated
Valid eprime	Recognizing right as right
Eprime violations	Recognizing wrong as wrong



In this chapter, we will see a bug I accidentally introduced in copying the test target and restyling it. We will also FIX it.



No matter what level of detail we look at when we test – code structures, application programming interfaces, user interfaces be it individual parts of multiple parts end-to-end – we should remember that with testing, we are not only evaluating our own code, but we are also evaluating our own code in the bigger ecosystem.

Quality experience of our users depends on things that are outside our direct control, and we may need to choose things in our direct control respectively to uphold promises we make for our users.

Signal – the messaging app – is a great example, and Naomi Wu makes the point for one aspect of quality (security) very clearly. You are only as secure as the weakest link in the system, and you can't have a secure application if it runs on a platform that isn't secure.

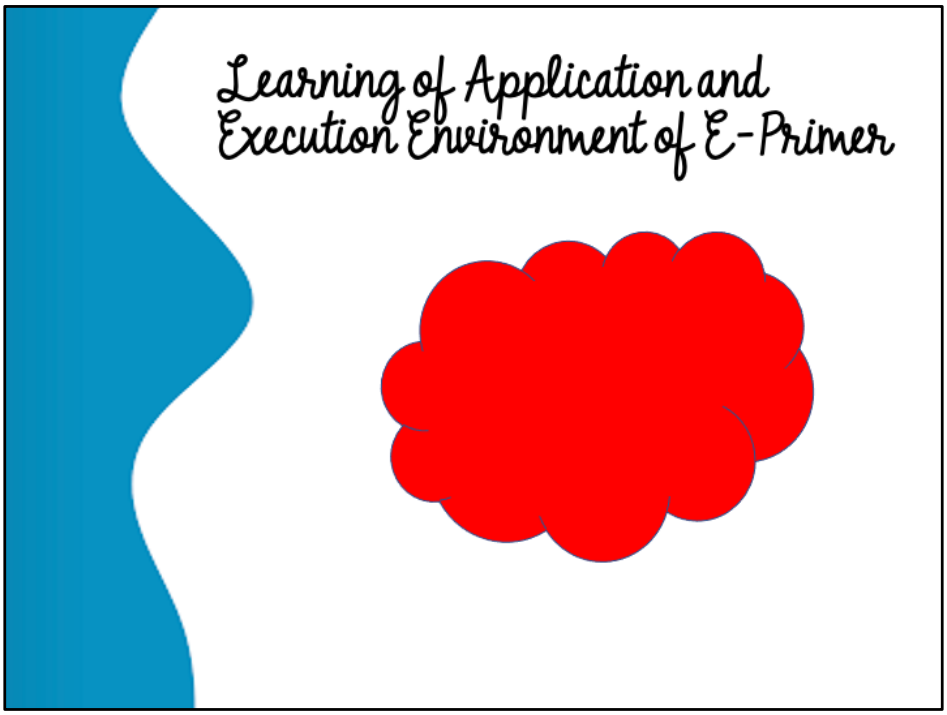
We evaluate applications as systems for multiple stakeholders.

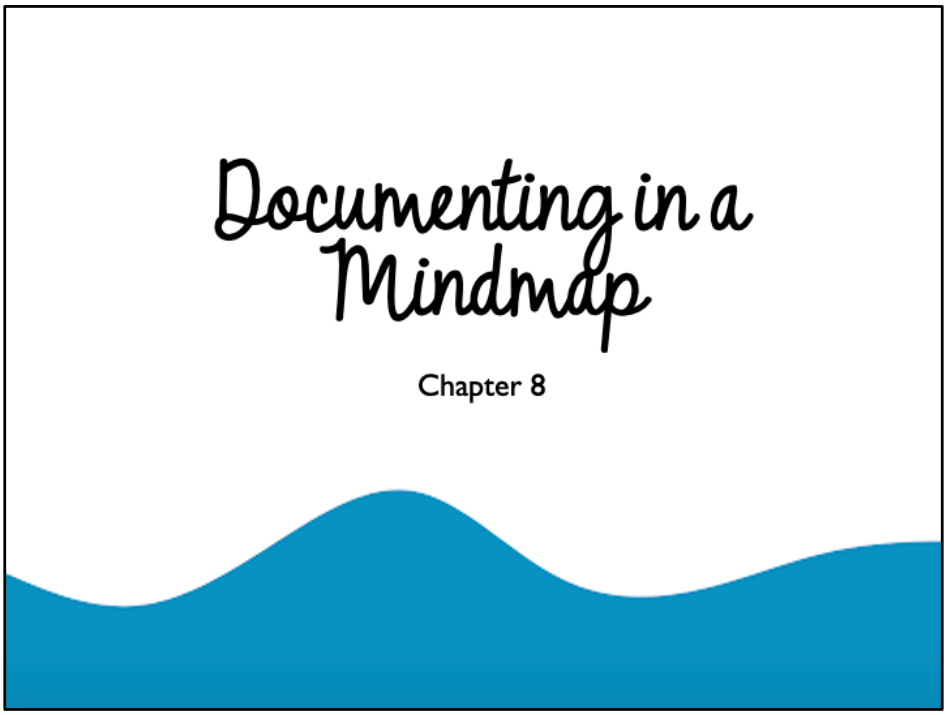
Execution Environment

Different browsers: web and mobile
Browser functionality and add-ons
HTML standard compatibility
Accessibility standard compatibility

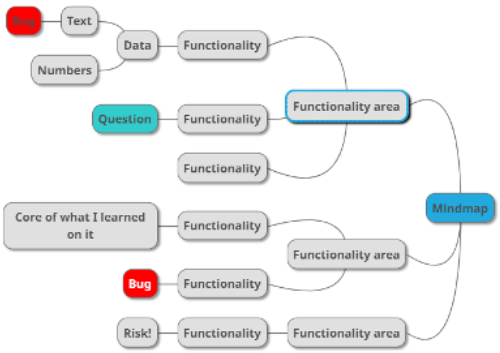
Let's Test

https://www.exploratorytestingacademy.com/app/
https://eviltester.github.io/TestingApp/apps/eprimer/eprimer.html


Learning of Application and
Execution Environment of E-Primer

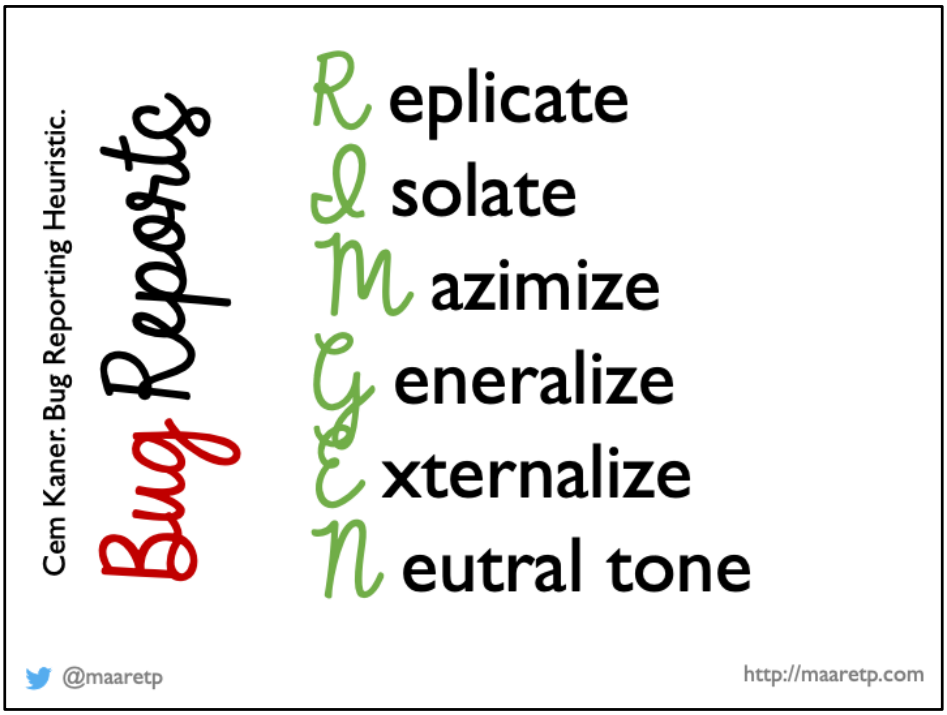

Documenting in a Mindmap
Chapter 8

Mindmap



@maaretp

http://maaretp.com


Cem Kaner. Bug Reporting Heuristic.
Bug Reports
Replicate
Isolate
Maximize
Generalize
Externalize
Neutral tone
@maaretp
http://maaretp.com

Let's Test

https://www.exploratorytestingacademy.com/app/
https://eviltester.github.io/TestingApp/apps/eprimer/eprimer.html

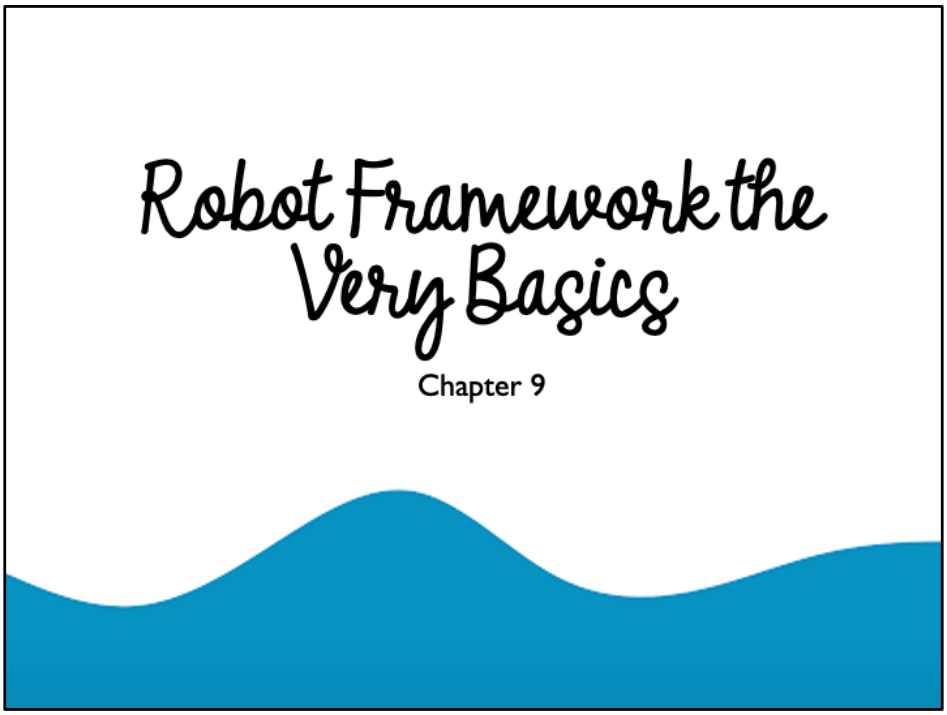
Mindmapping as Future Reference

Notetaking in the moment

Restructure as you learn

Documentation for the future

General purpose mindmaps


Robot Framework the Very Basics
Chapter 9

Robot Framework

Custom-made language
Built-in reporting
Ecosystem of keyword libraries

@maaretp

http://maaretp.com

Documenting as Skeleton Test Automation

Chapter 10

Log

```
1  *** Test Cases ***
2  This is a test case name
3      Log     First thing to do
4      Log     Second thing to do
5      Log     Third thing to do
```

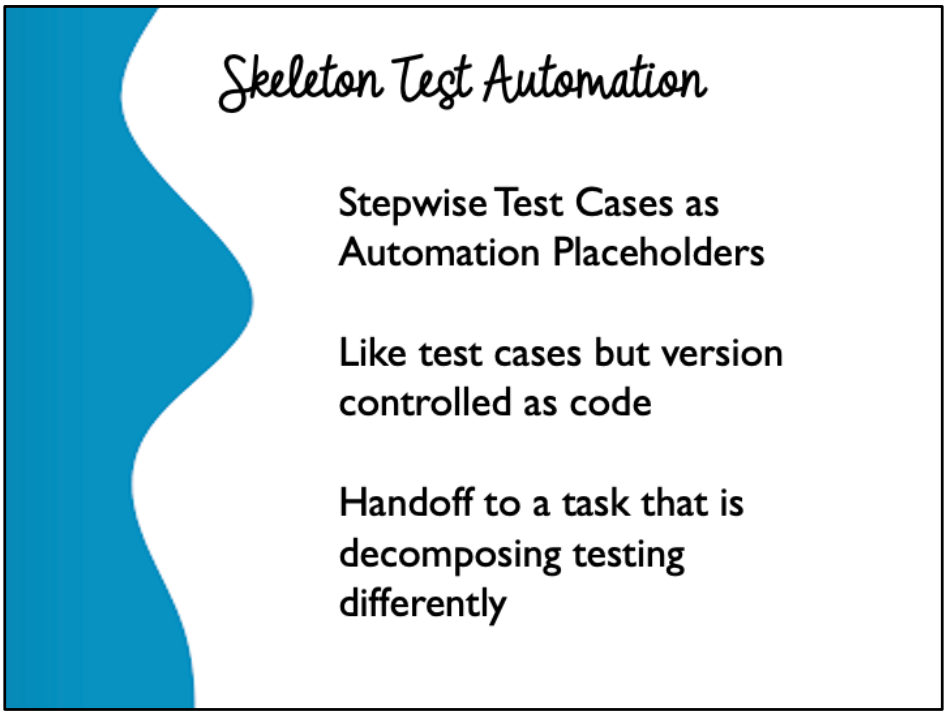
```
==============================================================================
Basic
==============================================================================
This is a test case name                                              | PASS |
------------------------------------------------------------------------------
Basic                                                                 | PASS |
1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed
==============================================================================
```

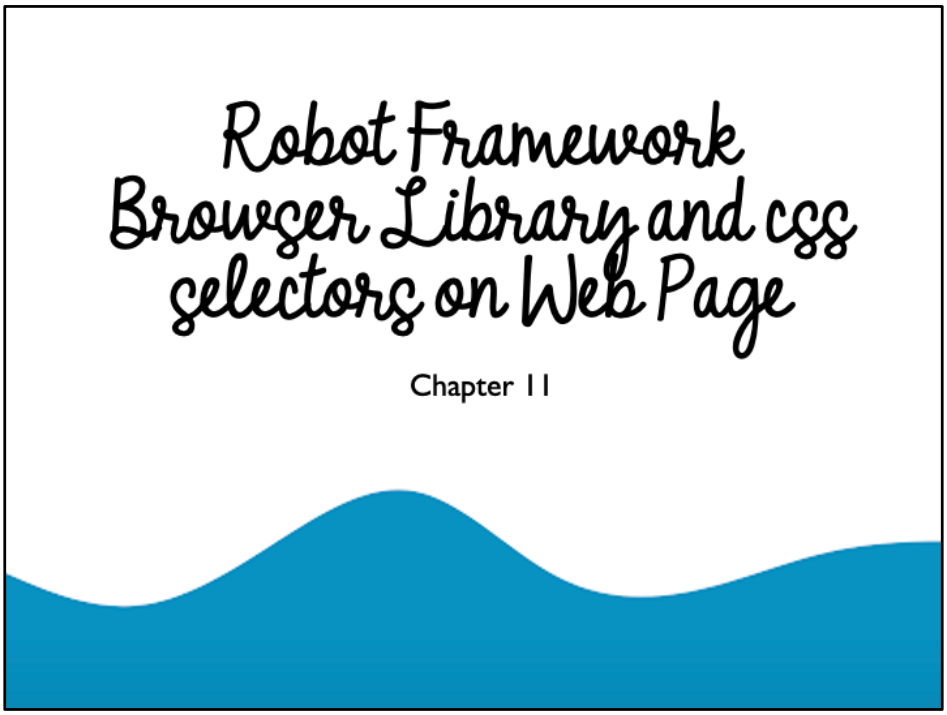
@maaretp

http://maaretp.com

Let's Test

https://www.exploratorytestingacademy.com/app/
https://eviltester.github.io/TestingApp/apps/eprimer/eprimer.html


Skeleton Test Automation
Stepwise Test Cases as Automation Placeholders
Like test cases but version controlled as code
Handoff to a task that is decomposing testing differently


Robot Framework
Browser Library and css
selectors on Web Page
Chapter 11

Browser Library

Playwright inside
Speed – Reliability – Visibility
Automatic waits

```
*** Settings ***
Library     Browser
```

@maaretp

http://maaretp.com

css selectors

css=
#id
.class
tag
[attribute='value']
[part_of_attribute_value_contains*='value']

@maaretp

http://maaretp.com

Keywords

```
*** Settings ***
Library     Browser

*** Test Cases ***
Open the Page Headless
    New Page    https://www.exploratorytestingacademy.com/app/
```

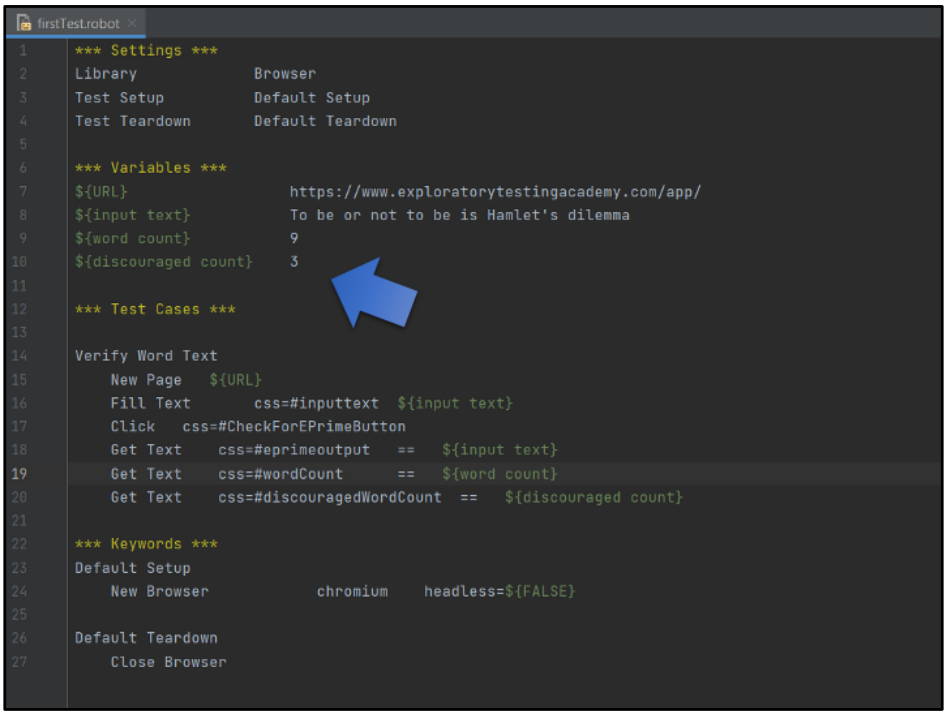
https://marketsquare.github.io/robotframework-browser/Browser.html

Documenting as Executable Test Automation

Chapter 12

Let's Test

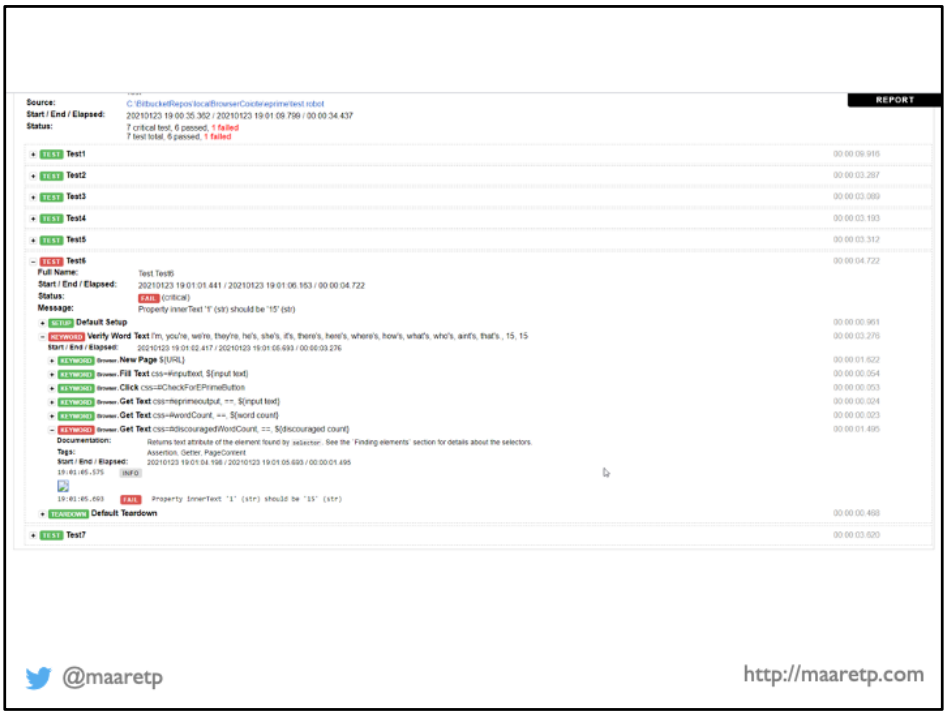
https://www.exploratorytestingacademy.com/app/
https://eviltester.github.io/TestingApp/apps/eprimer/eprimer.html


firstTest.robot
*** Settings ***
Library Browser
Test Setup Default Setup
Test Teardown Default Teardown
*** Variables ***
${URL} https://www.exploratorytestingacademy.com/app/
${input text} To be or not to be is Hamlet's dilemma
${word count} 9
${discouraged count} 3
*** Test Cases ***
Verify Word Text
New Page ${URL}
Fill Text css=#inputtext ${input text}
Click css=#CheckForEPrimeButton
Get Text css=#eprimeoutput == ${input text}
Get Text css=#wordCount == ${word count}
Get Text css=#discouragedWordCount == ${discouraged count}
*** Keywords ***
Default Setup
New Browser chromium headless=${FALSE}
Default Teardown
Close Browser


REPORT
firstTest Log
Generated
20210202 21:16:15 UTC+02:00
22 seconds ago
Test Statistics
Total Statistics | Total | Pass | Fail | Elapsed | Pass / Fail
Critical Tests | 1 | 1 | 0 | 00:00:03
All Tests | 1 | 1 | 0 | 00:00:03
Statistics by Tag | Total | Pass | Fail | Elapsed | Pass / Fail
No Tags
Statistics by Suite | Total | Pass | Fail | Elapsed | Pass / Fail
firstTest | 1 | 1 | 0 | 00:00:04
Test Execution Log
SUITE firstTest
Full Name: firstTest
Source: C:\BitbucketRepos\localBrowserCode\eprime\firstTest.robot
Start / End / Elapsed: 20210202 21:16:11.775 / 20210202 21:16:15.298 / 00:00:03.523
Status: 1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed
00:00:03.523
TEST Verify Word Text
Full Name: firstTest.Verify Word Text
Start / End / Elapsed: 20210202 21:16:12.650 / 20210202 21:16:15.292 / 00:00:02.642
Status: PASS (critical)
00:00:02.642
SETUP Default Setup 00:00:00.613
KEYWORD Browser New Page ${URL} 00:00:01.703
KEYWORD Browser Fill Text css=#inputText, ${input text} 00:00:00.036
KEYWORD Browser Click css=#CheckForEPrimeButton 00:00:00.049
KEYWORD Browser Get Text css=#eprimeoutput, ==, ${input text} 00:00:00.026
KEYWORD Browser Get Text css=#wordCount, ==, ${word count} 00:00:00.015
KEYWORD Browser Get Text css=#discouragedWordCount, ==, ${discouraged count} 00:00:00.014
TEARDOWN Default Teardown 00:00:00.160
@maaretp
http://maaretp.com

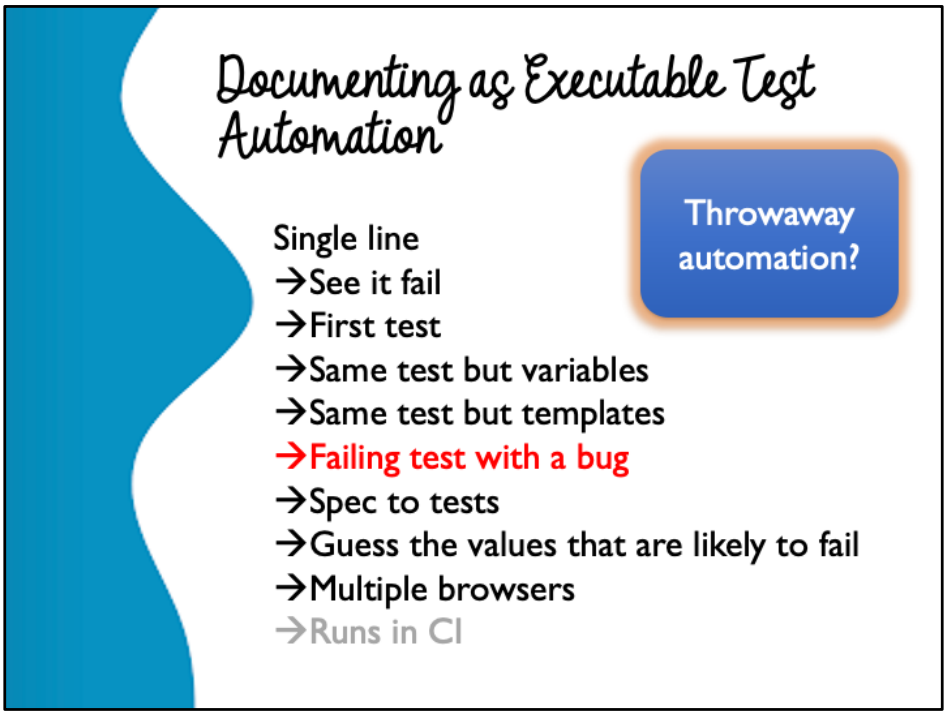
test.robot

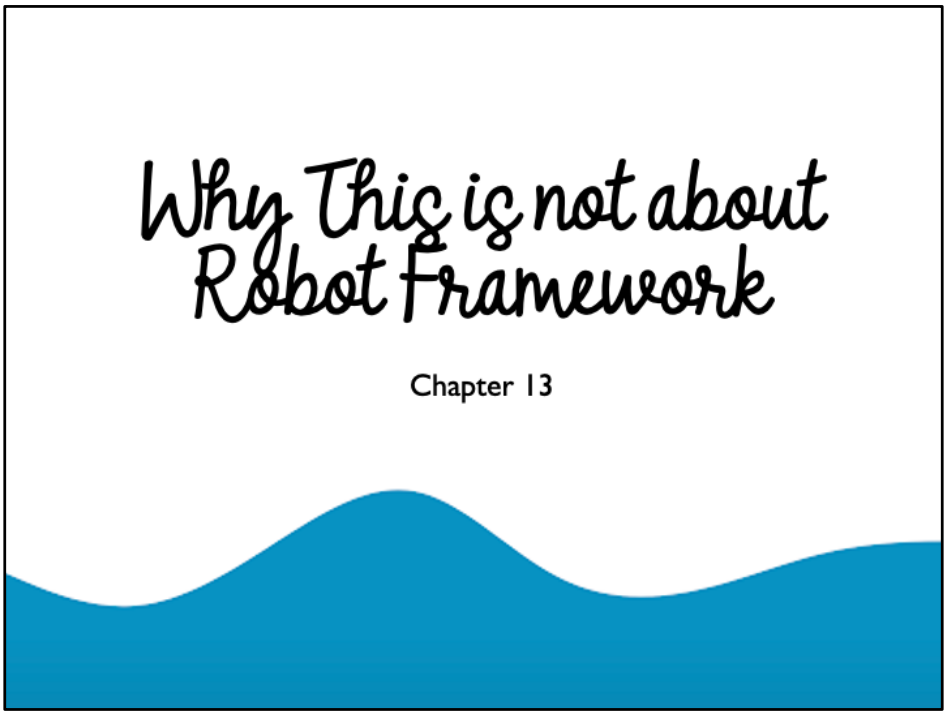
```
*** Settings ***
Library     Browser
Test Setup          Default Setup
Test Teardown       Default Teardown
Test Template       Verify Word Text

*** Variables ***
${URL}      https://www.exploratorytestingacademy.com/app/

*** Test Cases ***
Test1    nothing    1    0
Test2    to be or not to be    6    2
Test3    The cat is my only pet    6    1
Test4    The cat is Garfield    4    1
Test5    be, being, been, am, is, isn't, are, aren't, was, wasn't, were, and weren't.    13    12
Test6    I'm, you're, we're, they're, he's, she's, it's, there's, here's, where's, how's, what's, who's, aint's, that's.    15    15
Test7    ${EMPTY}    0    0

*** Keywords ***
Verify Word Text
    [Arguments]     ${input text}      ${word count}      ${discouraged count}
    New Page   ${URL}
    Fill Text       css=#inputtext  ${input text}
    Click   css=#CheckForEPrimeButton
    Get Text    css=#eprimeoutput   ==   ${input text}
    Get Text    css=#wordCount      ==   ${word count}
    Get Text    css=#discouragedWordCount  ==   ${discouraged count}

Default Setup
    New Browser           chromium    headless=${FALSE}

Default Teardown
    Close Browser
```

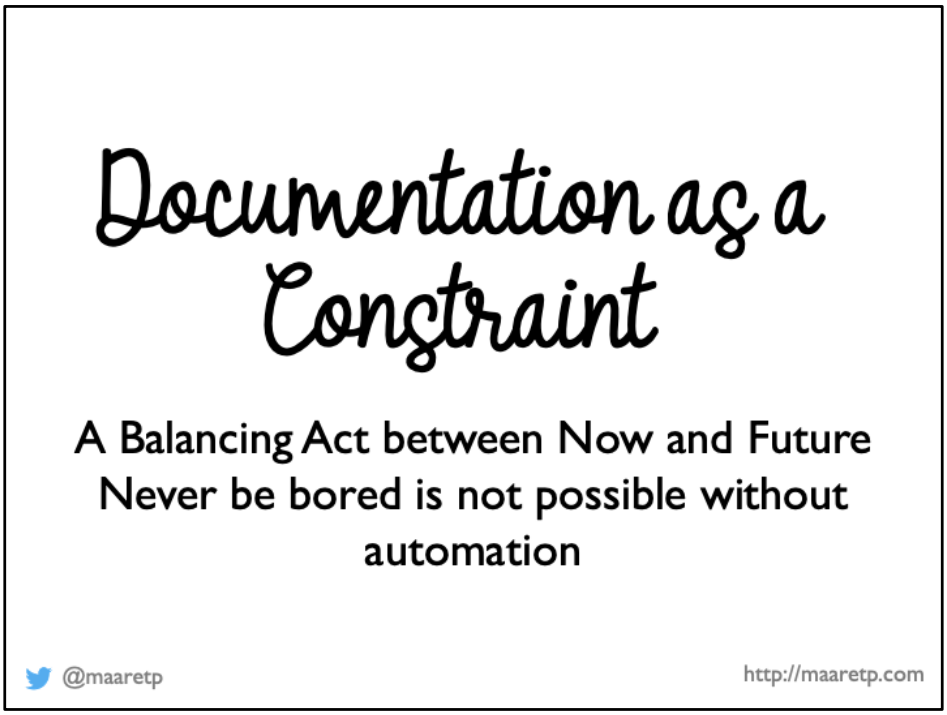


@maaretp

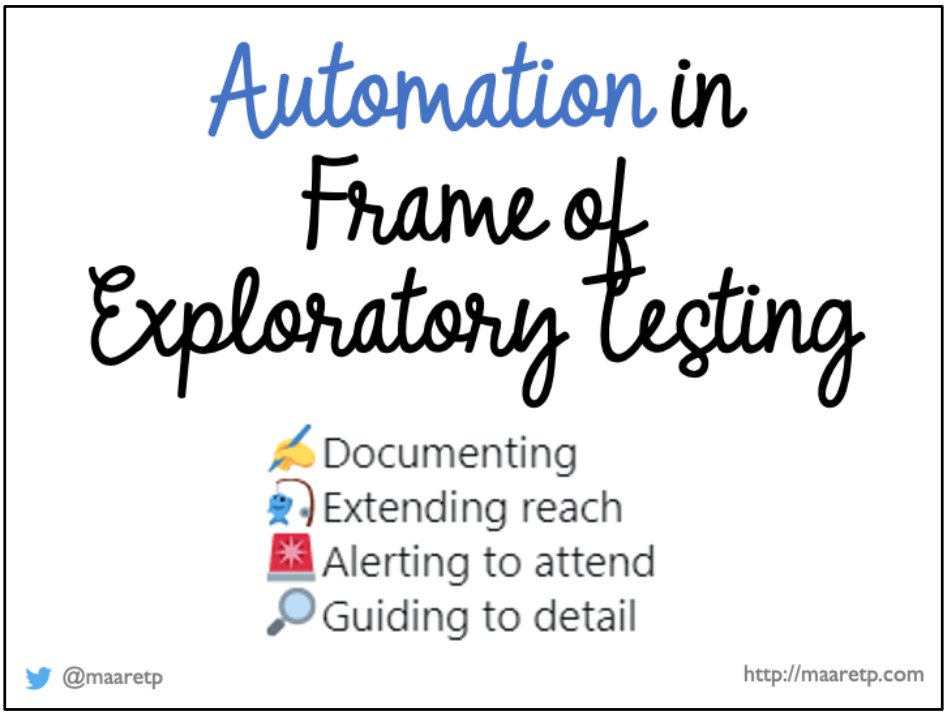
http://maaretp.com


Documenting as Executable Test Automation
Throwaway automation?
Single line
→See it fail
→First test
→Same test but variables
→Same test but templates
→Failing test with a bug
→Spec to tests
→Guess the values that are likely to fail
→Multiple browsers
→Runs in CI

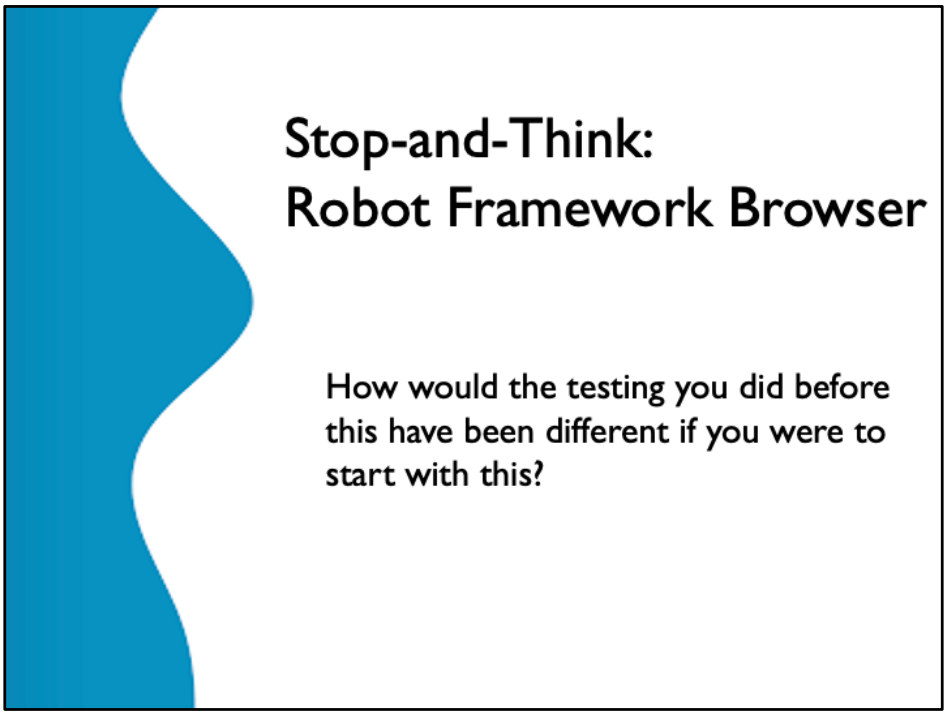

Why This is not about
Robot Framework
Chapter 13

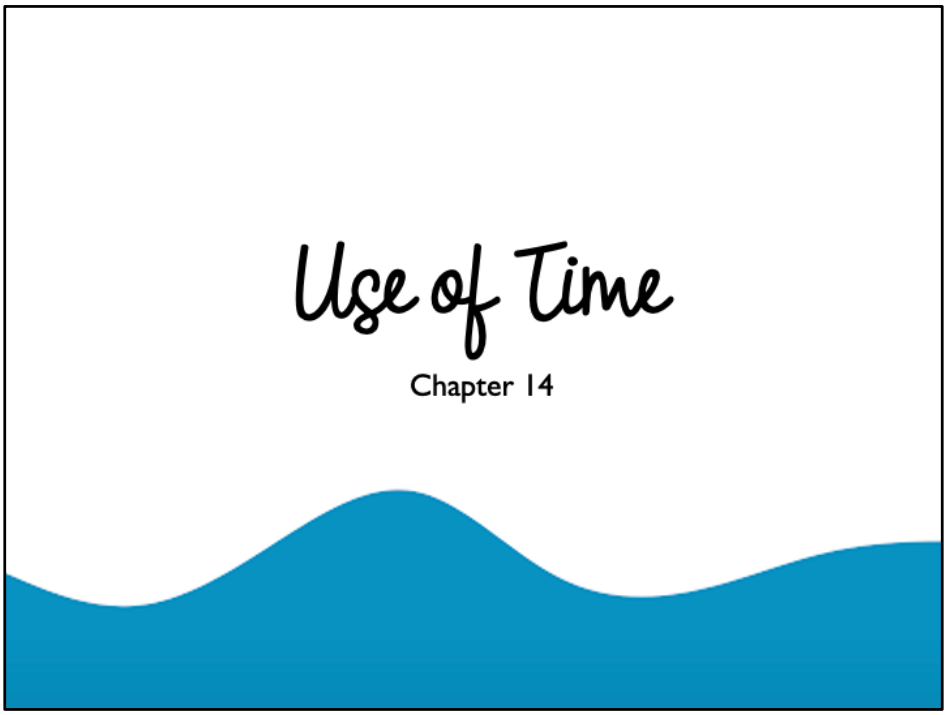


Testers cope with tedium, developers automate it away. Best of both worlds!



Improving testability

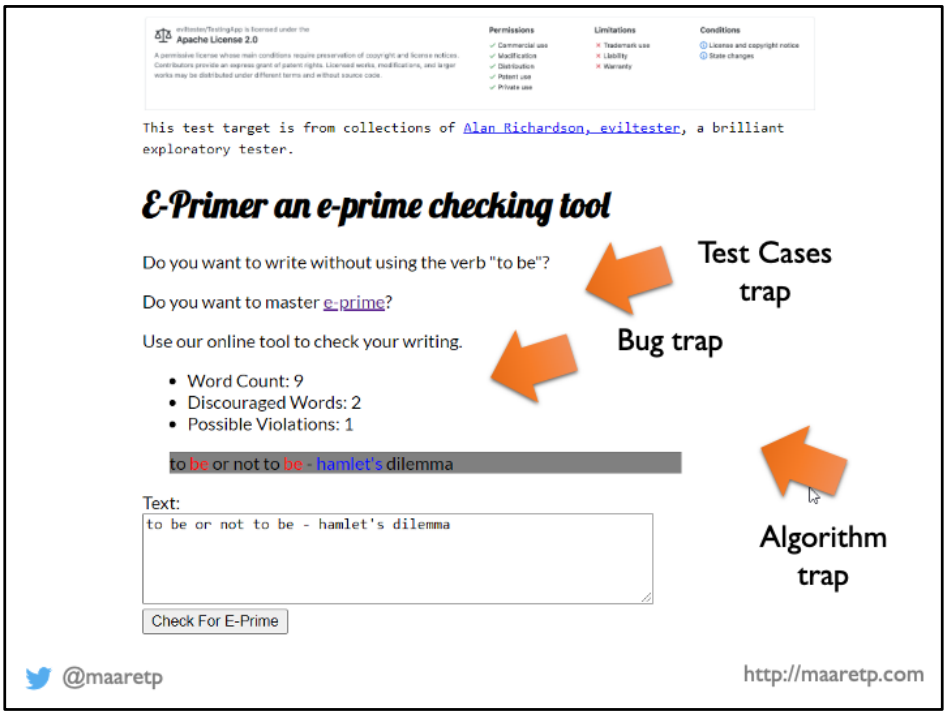

Stop-and-Think:
Robot Framework Browser
How would the testing you did before this have been different if you were to start with this?


Use of Time
Chapter 14

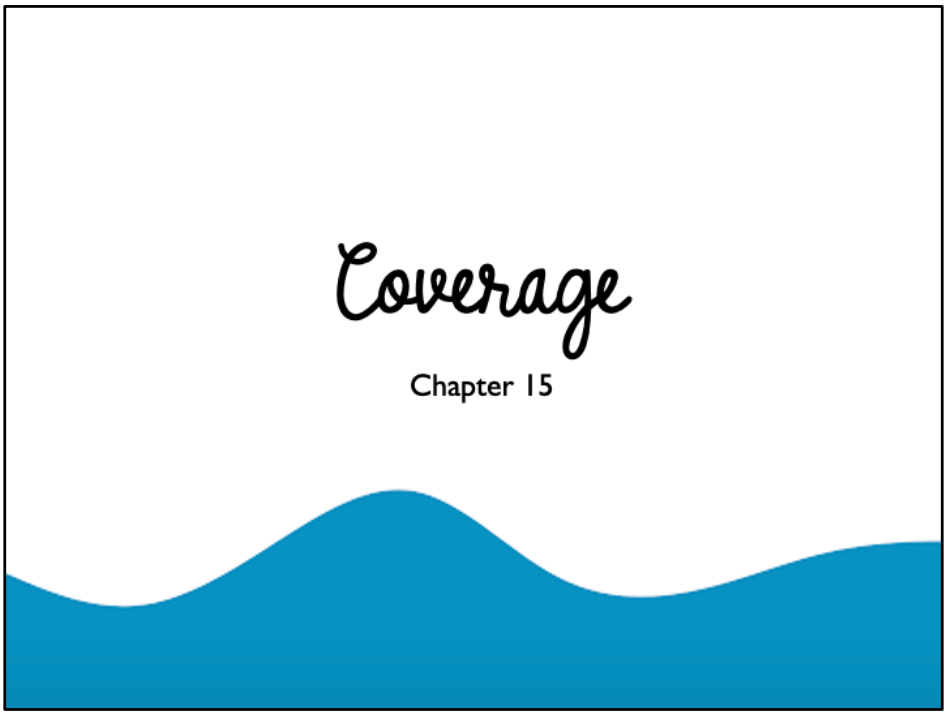
Test, Bug, Setup

Software with little bugs is faster to test
Setup is configuring, learning and documenting
Test grows coverage

@maaretp http://maaretp.com

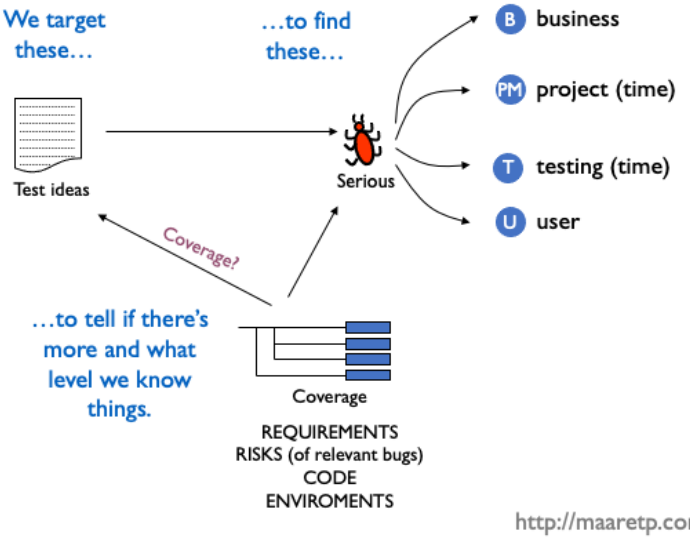

Apache License 2.0
Permissions
Limitations
Conditions
This test target is from collections of Alan Richardson, eviltester, a brilliant exploratory tester.
E-Primer an e-prime checking tool
Do you want to write without using the verb "to be"?
Do you want to master e-prime?
Use our online tool to check your writing.
Test Cases trap
Bug trap
Algorithm trap
Word Count: 9
Discouraged Words: 2
Possible Violations: 1
to be or not to be - hamlet's dilemma
Text:
to be or not to be - hamlet's dilemma
Check For E-Prime
@maaretp
http://maaretp.com


Stop-and-Think:
Time and Traps
Where did your time go on testing of the application?


Coverage
Chapter 15

Setting the Stage for Testing

WHAT
WHEN
WHO
HOW
WHY



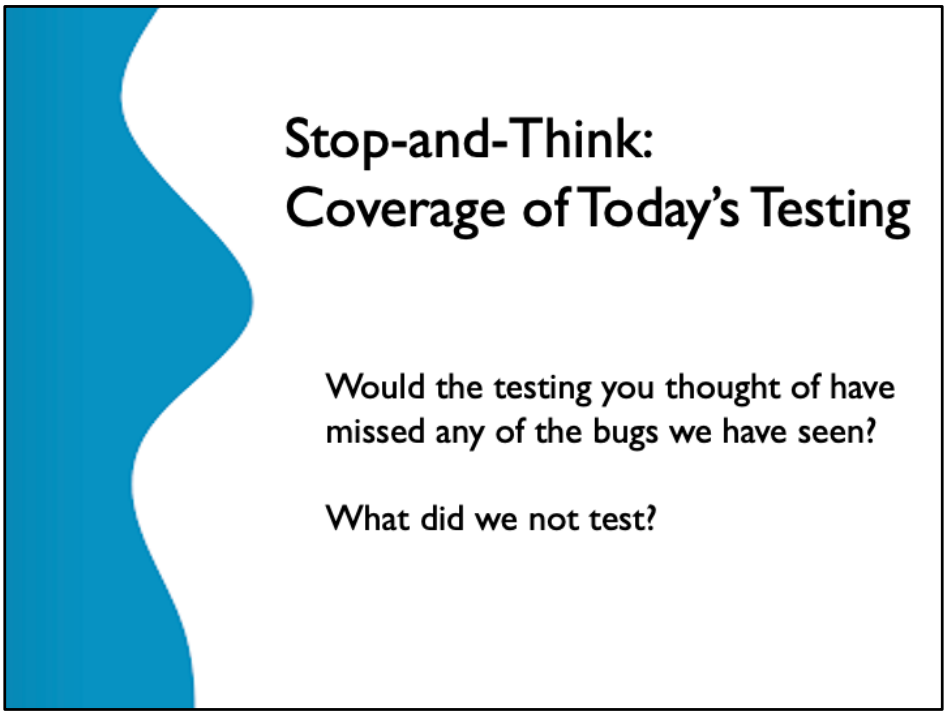
@maaretp

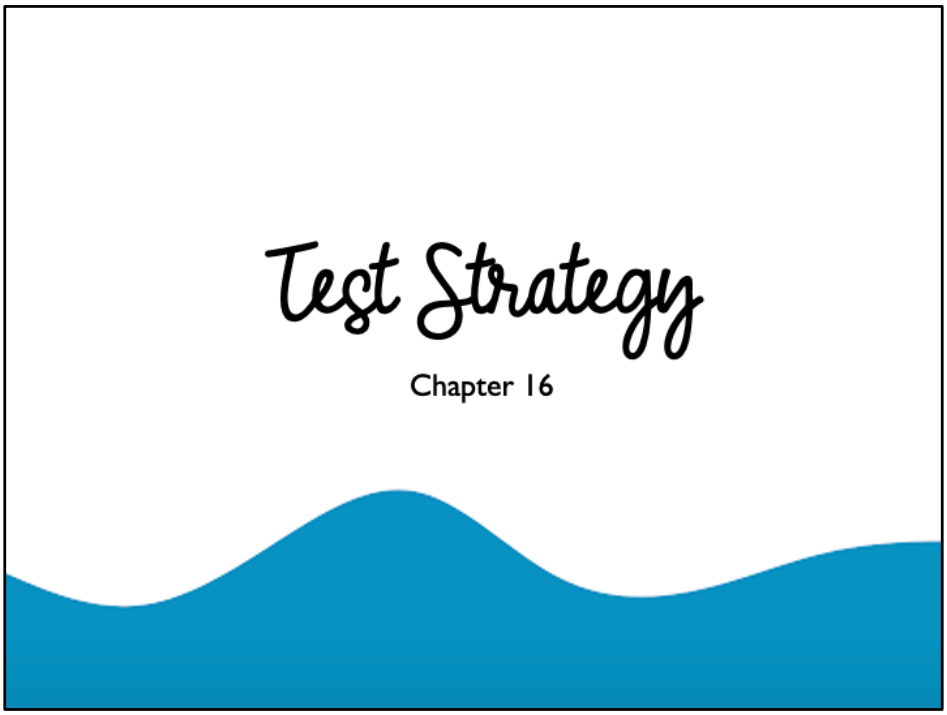
http://maaretp.com

Risk Coverage

Coverage of relevant bugs
Effectiveness – results of overall strategy facilitate experience of quality for stakeholders

@maaretp http://maaretp.com

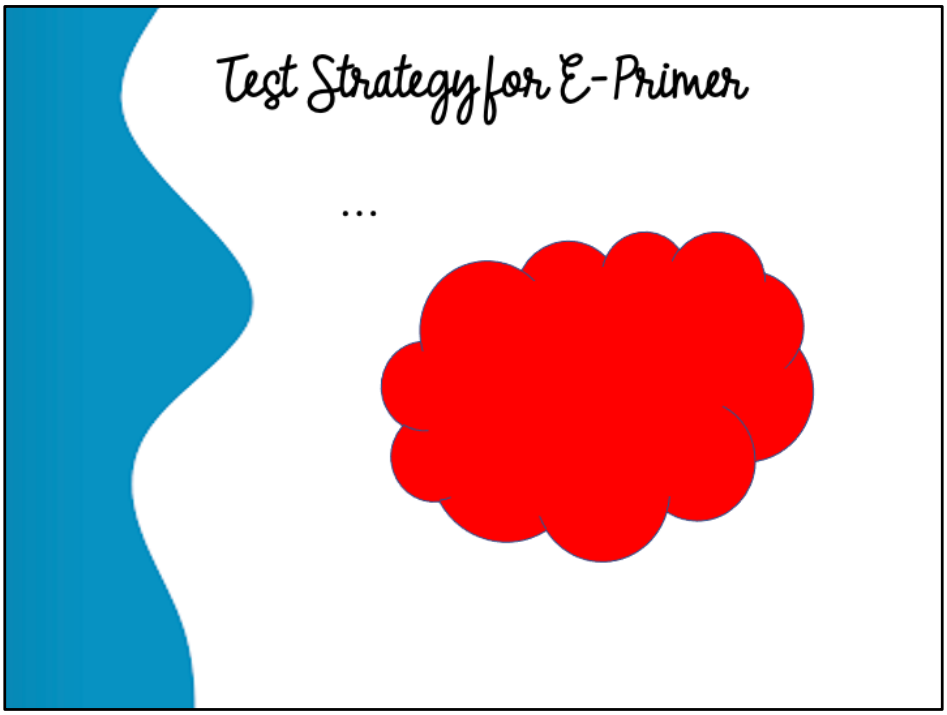

Stop-and-Think:
Coverage of Today's Testing
Would the testing you thought of have missed any of the bugs we have seen?
What did we not test?

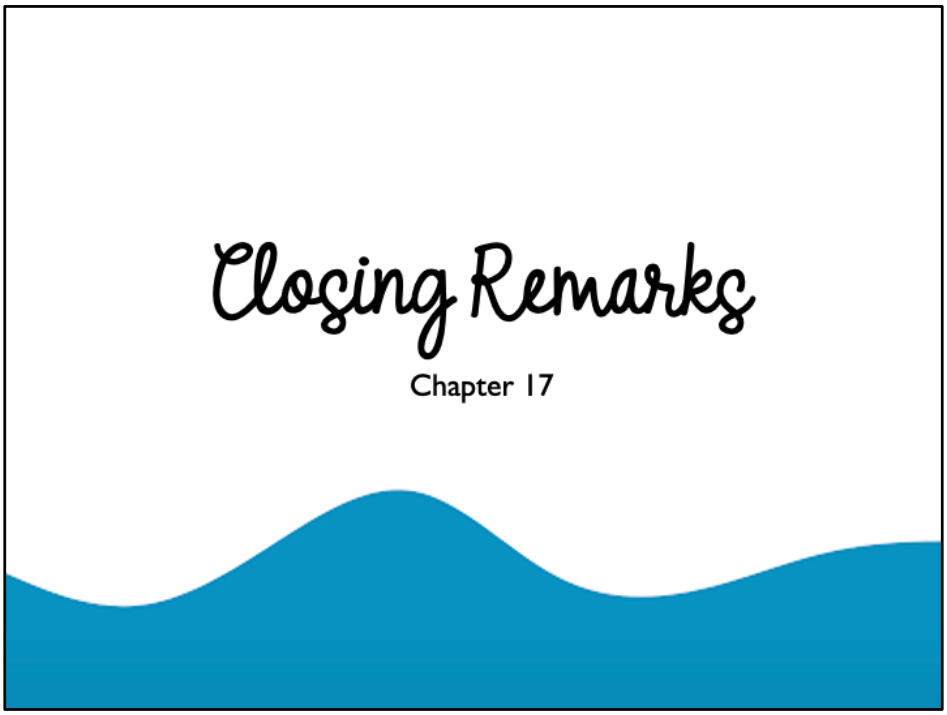

Test Strategy
Chapter 16

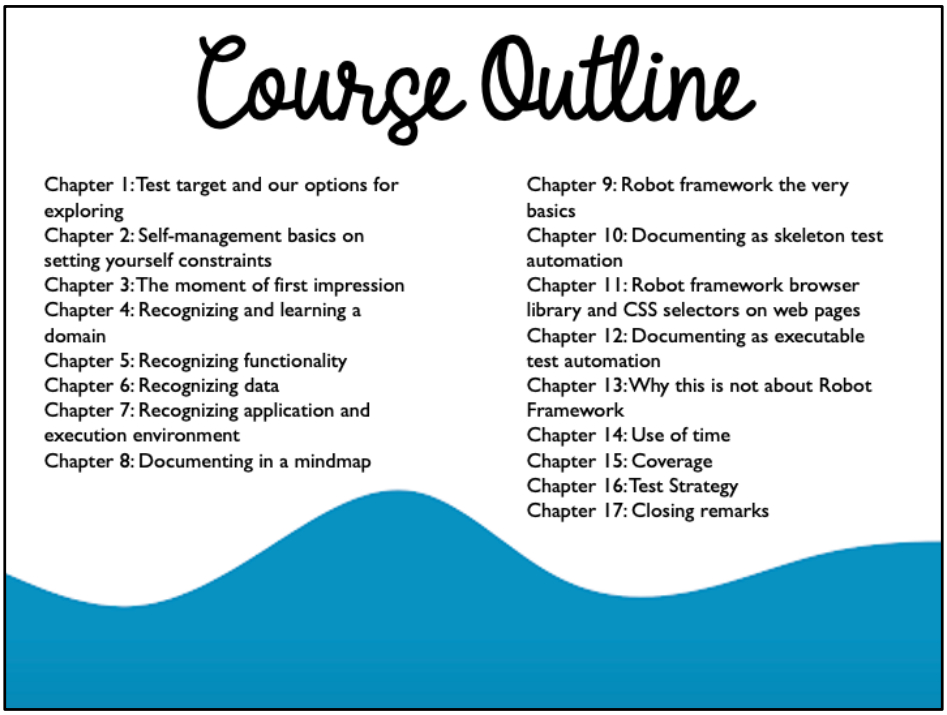

Ideas that Guide Test Design
Specific to Application Under Test
Risks to ways of testing for them
@maaretp
http://maaretp.com

Let's Test

https://www.exploratorytestingacademy.com/app/
https://eviltester.github.io/TestingApp/apps/eprimer/eprimer.html


Test Strategy for E-Primer
...


Closing Remarks
Chapter 17



This course is first in the series of Contemporary Exploratory Testing courses where teaching on the course is framed around a particular test target. For foundations, our application is very simple, and it was chosen based on the idea that is *not supposed to be full of bugs*. While finding lots of bugs is fun and does wonders to a tester's self-esteem, it gives a false impression of what it is we do when we do exploratory testing.

The course is split into 17 chapters, where each chapter will have a supporting video, and the final chapters will describe the best testing we have done on the application. It isn't a recipe for all applications, but an example for you to understand what coverage for this application may mean, and what parts of the work you could do make sense when you are truly optimizing value.

From the outline, you can see that we will be using a test automation framework (Robot Framework) on this course. Choice of tool is irrelevant but using one to give you concrete examples helps teach this content. Robot Framework allows for very natural language like programming in its own custom-made language and can be useful for new people to get into test automation programming, even if I believe it soon becomes a limiting factor you may want to learn away from.

Maaret Pyhäjärvi

EuroSTAR
TESTING EXCELLENCE
AWARD
2020

AGILE
TESTING DAYS
MIATPP
Most Influential Agile Testing
Professional Person
2016

Email: maaret@iki.fi
Twitter: @maaretp
Web: maaretp.com
Blog: visible-quality.blogspot.fi
(please connect with me through Twitter or LinkedIn)

#PayToSpeak #TechVoices
#EnsembleTesting #EnsembleProgramming #StrongStylePairing
#ExploratoryTesting #TestAutomation
#ModernAgile
#AwesomeTesters