recheck and the “Git for the GUI”

Dr. Jeremias Rößler

roessler@retest.de

Abstract

Ever had that: after a simple change, suddenly 50+ tests are failing! Brittle tests that hinge on GUI specifics and result in the dreaded `NoSuchElementException` are a main headache when testing with Selenium. How about a tool that does not depend on HTML-IDs, CSS classes or XPath? Since they are invisible, they are essentially irrelevant from a user’s perspective—yet they are crucial for the test to succeed.
The open source project *recheck* offers a simple and elegant solution. Not only is a virtual identifier unaffected by UI changes, you can define it for otherwise hard to specify elements, i.e. that would require complex XPath or CSS selector expressions. And on top of that, tests are easier to create and maintain and yet much more complete in what they check.

Biography

Dr. Jeremias Rößler (Roessler, @roesslerj, he) has a PhD in Computer Science from Saarland University and more than 10 years of experience as a software developer and tester. He is the founder and CEO of @retest_en (https://retest.de), a German-based startup that brings AI to test automation and one of the authors of the iSQI/GASQ "AI and Software Testing" certification syllabus.

His refreshingly unusual approach to test automation (difference testing) has many advantages over conventional test automation and he shows how to combine it with AI to overcome the oracle problem. He has been speaker at many international conferences, both in academia and industry, and attendees call his talks visionary and amusing. His talks are rated 4.28 out of five and ranked second best of the conference. He is a writer, blogger (https://dev.to/roesslerj/), developer & computer scientist.

He is very approachable and enjoys to be talked to, so don’t be shy. You can contact him easiest on Twitter or LinkedIn (https://www.linkedin.com/in/roesslerj/).

Copyright Jeremias Roessler, August 2020

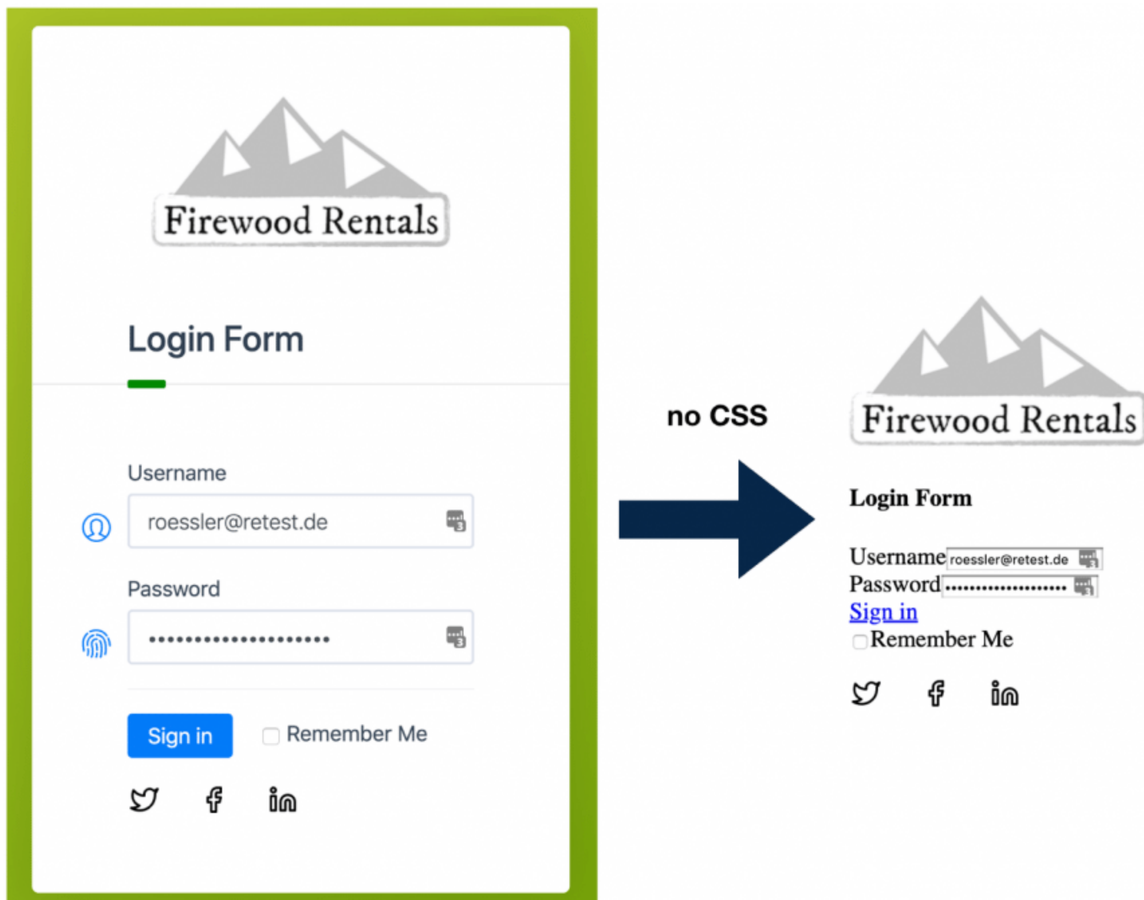
1 The Problem

In software development and test automation, assertion-based testing is the classic JUnit approach, where manually created assertion statements serve as a test oracle (to distinguish between pass/fail) during test execution. Essentially, they check the result of a calculation—usually by comparing it with a manually defined expected value. For unit-based test automation (i.e. testing the system from within), this is very well suited. But applying it to testing an interface (specifically the user interface) has proven to be problematic, as this article will explain. Assertion-based Testing is a deny-list approach, i.e. only changes of the software that are explicitly checked by assertions are recognized and alerted (denied).
Selenium is a very good tool for web-based test automation. It doesn't promote assertion-based testing itself—however, most people use it in conjunction with an assertion-based checking library. To test a web application login, a typical Junit-style Selenium test could look something like the following. All examples in this article are available on GitHub (https://github.com/retest/recheck-web-example).

```
class MySeleniumTest {

   WebDriver driver;

   @BeforeEach
   void setup() {
      driver = new ChromeDriver();
   }

   @Test
   void login() throws Exception {
      String file = "src/test/resources/demo-app.html";
      driver.get(Paths.get( file ).toUri().toURL().toString());

      driver.findElement(By.id("username")).sendKeys("Simon");
      driver.findElement(By.id("password")).sendKeys("secret");
      driver.findElement(By.id("sign-in")).click();

      assertEquals(driver.findElement(By.tagName("h4")).getText(),
            "Success!");
   }

   @AfterEach
   void tearDown() throws InterruptedException {
      driver.quit();
   }
}
```

As you can see, it opens a login page, fills username and password and clicks "sign-in". Then it checks whether the text "Success!" is in the first h4 header.

Now you might want to change the HTML of the website under test. For example, you could change the CSS declaration `<link href="./files/main.css" rel="stylesheet">`. Changing a single character of the URL will cause the website to be displayed without formatting.



This change obviously is an error. However, when you run the test, it shows no problem and still executes without failure. This is clearly not what you expect from the test. Instead, try to change or remove the element IDs that are invisible to the user. Since these IDs are not visible, the change has no effect on the actual website from the user's point of view. But if you run the test now, you can see that it terminates with a `NoSuchElementException`. The change, which is irrelevant to the user, not only caused the test to fail, but also prevented its execution—in other words, "broke" it. Tests that ignore major changes but break when changes are invisible are the current standard in test automation. This is just the opposite of what you would want from a test.

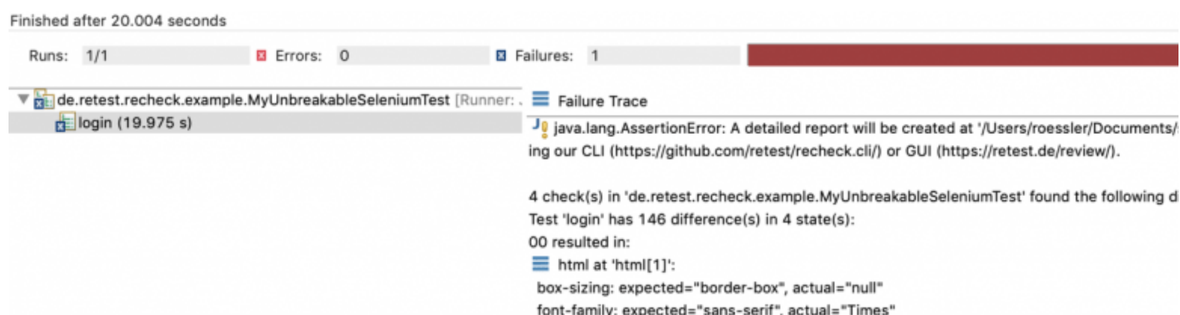
2 A different Approach

Difference Testing, in contrast, is a Golden Master-based testing approach (also called characterization testing and approval testing) that highlights the differences between different executions, such as between different versions of the software. With Difference Testing, all differences (including unexpected ones) are detected, and irrelevant ones can then be ignored. Hence Difference Testing is an allow-list approach, allowing certain changes to happen without alerting, while alerting for all changes that are not explicitly ignored. Again, this is essentially the opposite of an assertion-based testing approach, where you explicitly specify what should be checked—and *all other changes* are ignored.

The number of tools that apply an Golden Master-based approach to testing—such as Approval Tests, Jest, or recheck-web (retest)—seems to be constantly increasing. This approach promises tests with less effort (for both creation and maintenance) while testing more thoroughly. And in the case of recheck-web, the tests are even more robust (as will be shown below).
For our above example, all we need to change is wrapping the Selenium driver with a `RecheckDriver`:

```
driver = new RecheckDriver(new ChromeDriver());
```

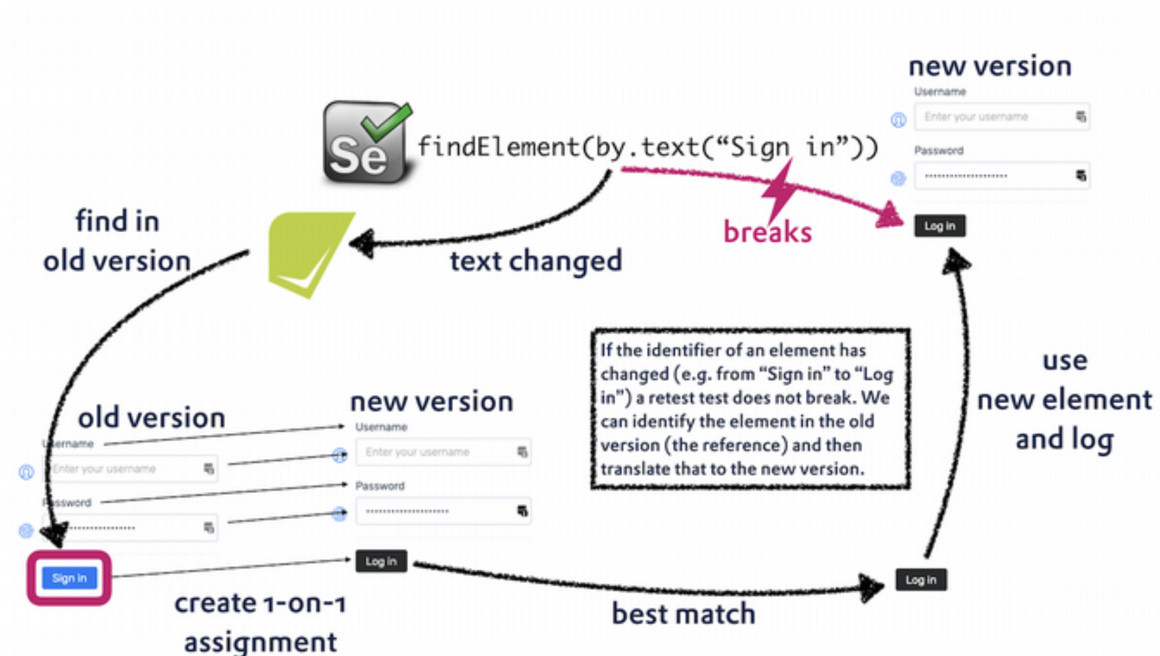
With a correct project configuration, nothing else is necessary. You can now delete the assertion-based check. If you execute this test the first time, the Golden Master (the reference which to compare against) hasn't been created yet. So for safety, the first time you execute this test, it will fail and create said Golden Master. If you remove the CSS from the website with the above change after the Golden Master has been created, the test suddenly fails with many differences.



But what is much more interesting: If developers change or remove the HTML element IDs instead, the test is still executed successfully (works accordingly with name, CSS class, XPath etc.). Remember, these element IDs are being used to identify the element the test should interact with. Yet still without these IDs on the website, the test still doesn't brake—although it correctly reports these IDs as having changed. This is how a test should behave: It should detect changes that are important to the user and not break on changes that are irrelevant to the user.

3 An Open Source Implementation

recheck-web is a free open source (https://github.com/retest/recheck-web) tool based on Selenium. It works according to the Golden Master principle, which essentially means that the first time the test is run, it creates a copy of the rendered website, and subsequent runs of the test compare the current state with this copy (the Golden Master). This allows the test to detect whether the website has changed in any undesirable way. After changing or deleting an HTML ID on the website to test, recheck-web can still identify the corresponding element by simply looking into the Golden Master (where the ID still exists) and find the element there. Using additional properties like HTML name, XPath and CSS classes, recheck-web can still correctly identify the element on the changed website and return it to Selenium. The change is reported and the test can then interact with the element as before.



The idea to do testing Golden Master-based exists since probably the 80s. It has been made popular under the term "characterization testing" by Michael Feathers with his book "Working Effectively with Legacy Code". The reason for the approach to be not more widespread is probably based on two major disadvantages: noise and redundancies.

3.1 Noise and how to address it

Many changes are uninteresting and unproblematic (e.g. time and date changes and random IDs as well as many uninteresting style attributes). As was explained above, while assertion-based checking ignores *all* changes unless explicitly specified, Golden Master-based checking ignores *no* changes unless explicitly specified. Following that reasoning, one could argue that assertion-based checking is essentially creating and maintaining a deny-list of changes, while Golden Master-based testing is creating and maintaining an allow-list of changes.
Since this is a major cause of effort, recheck goes out of its way to make it easy to create and maintain that allow-list. For the same reason that Git uses the `.gitignore` file to ignore log files and other temporary and uninteresting files and artifacts, recheck-web uses the `recheck.ignore`-file. After executing a recheck-web test the first time, that file is automatically created in the correct (default) directory. As with everything, this can be configured if needed. With a Git-like syntax, it's easy to edit that file and add additional elements, attributes or changes to ignore. You can even create different such files for different purposes and scenarios (then called filter files like `positioning.filter`) and mix and combined them as needed.
Every line in a filter file constitutes to an ignore rule. It essentially boils down to

```
<element-matcher> <attribute(s)> <specifics>
```


Copies may not be made or distributed for commercial use

An element matcher specifies which element and sub-elements the rule applies to. This can be done by type, HTML ID, CSS class, XPath, retestId or others. If there is no element matcher specified, it applies globally. Then is specified to which attribute(s) the rule applies, by name or regex. If no attributes are specified, the element and all of its child elements are ignored completely. At last, some specifics can be defined, making the rule apply only for certain values or differences. This allows e.g. to ignore a text, as long as it's a valid date, or a pixel-difference as long as it's less than 20 pixel. More details on that can be found online (https://docs.retest.de/recheck/usage/filter/).
The remarkable thing with the `recheck.ignore` file is, that it usually works project-wide. This means that you need to maintain just a single file for the whole project. And adding the next test usually means less effort in terms of the recheck.ignore file than the previous one was, until at one point, it becomes completely effort-less.

3.2 Redundancies and what to do with them

The second reason that Golden Master-tests can be cumbersome to maintain is redundancies. Several Golden Masters often have a certain, sometimes even a high overlap. Then the same change has to be checked and confirmed several times, which quickly nulls the efficiency achieved with the simpler test creation.
To counter that problem, recheck-web comes with its own CLI (https://github.com/retest/recheck.cli) that takes care of this tedious task. With it (or the commercial GUI) users can apply the same change for the same element to all Golden Masters or simply globally apply or ignore all changes with a single or very few commands, like

```
recheck commit --all test.report
```

If you want to apply a change so a subset of elements, then the filters come in handy again. You can either specify a filter file that should be used to select the changes, or you can define a filter impromptu on the command line.
It is very important to maintain the Golden Master and regularly apply the changes. If not done, it means that over time the Golden Master and the actual output continue to diverge. At some point in time, the differences are to stark, so that the algorithm that assigns the individual elements from the Golden Master to the elements from the current version gets tricked, and will start to make mistakes. So using the CLI or GUI is mandatory in the long run. However, it also means that tests that were kept from becoming broken thanks to the Golden Master as explained above, will then break once the changes are applied. For instance, in the example above with a changed HTML ID, once that changed or deleted HTML ID gets applied to the Golden Master, the outlined mechanism will fail and the test will break. What we can use instead of the ordinary HTML ID, or any other single identification criteria for that matter, is using the `retestId`.

4 The retestId

The idea of the `retestId` is fairly simple: when creating the copy of the website (known as the Golden Master), we can insert additional attributes (say an ID) to each element inside of that copy. Since these attributes only reside in the copy, not on the actual website, they can never be subjected to changes. So if we ensure that these attribute values are unique, it means we just

created a virtual constant identifier. This virtual constant identifier is added to every element and is called the `retestId`. The overall idea works essentially like outlined above: we find the element by its retestId in the Golden Master and then do a 1-on-1 assignment of all elements of the old to the new website. Then we use the element from the current website that has been assigned the retestId and return it to the underlying Selenium.
In addition to being constant, the retestId addresses another problem that is common in test automation: often enough, the element that one wants to addresse doesn't have a single unique identification criterion like an HTML ID, name, or CSS class. In that case, one is usually forced to use a XPath or CSS path as identification criterion, both being meaningless and hard-to-maintain. With the retestId, you can instead use a meaningful and short String that can be manually set to any value, as long as it's unique within the Golden Master.

5 Outlook

The reason recheck was created and is made open source is simple: It addresses one of the main challenges of a really interesting problem: The Oracle problem in AI-based UI-test generation. The Oracle problem is the very hard problem to predict whether a certain outcome of a software after a certain input is correct. Since problems are in the eye of the beholder, this problem cannot be solved in a general way. So instead, recheck circumvents the problem: the software gets manually tested *once*. After the outcome is determined to be correct *enough*, it gets frozen in the form of the Golden Master. From then on, only *improvements* are to be allowed—all other changes to the output should be rejected by fixing the underlying changes in the code (regressions). This then essentially means that test automation has become an extension of change control or version control, further tying the above association with e.g. git. Think about it this way: version control governs static artifacts like code and configuration files. The dynamic behavior of the software is unfortunately defined by more (data, runtime system)—thus we need automated tests to close this gap. The current state in software test automation however, is woefully inadequate and only provisionally addresses this need. Recheck on the other hand consequently follows this approach up to the `recheck.ignore` file and can be rightfully called "Git for the GUI".

6 Conclusion

In this paper, we have presented an open source tool that implements and demonstrates a different approach to software test automation than the current assertion-based one. We have also shown how well-known shortcomings of that approach can be addressed to make it applicable to real-world projects, and how it paths the way to AI-based test generation by addressing the dreaded Oracle problem.

References

Michael Feathers. 2004. *Working Effectively with Legacy Code*. Prentice Hall International. EAN / ISBN-13: 9780131177055

Approval Tests. https://approvaltests.com/. Accessed August 1, 2020.

Jest. Facebook Inc. https://jestjs.io/. Accessed August 1, 2020.

retest-web-example. ReTest GmbH. https://github.com/retest/recheck-web-example. Accessed August 1, 2020.

retest-web. ReTest GmbH. https://github.com/retest/recheck-web. Accessed August 1, 2020.

recheck documentation. ReTest GmbH. https://docs.retest.de/recheck/usage/. Accessed August 1, 2020.

recheck CLI. ReTest GmbH. https://github.com/retest/recheck.cli. Accessed August 1, 2020.