Fighting the repetition

Insights from a successful test automation project

Where we started

Testing has a bad reputation among developers, because it is often associated with dull and repetitive activities. Testing is associated with little room for creativity. Any deviation from a given procedure may trigger an unintended response or spoil the test result. Nobody really enjoys such tasks.

This document offers a quick overview how we at Thetaris automated a variety of tests in our projects. The main goal was to free developers from repetitive tasks and improve developer productivity. To achieve this goal, we made intensive use of available tools. In some places we added our own slant of solutions. Recent technologies from fields as diverse as image recognition, visualization and artificial intelligence allowed us to come up with some new ways for automating tests.

Our mantra for the test automation consisted of three don'ts that we wanted to avoid:

- Don't waste time between failure and fix
- Don't waste time finding the most suitable developer for the fix
- Don't waste time setting up new test cases

For our testing framework this had three clear consequences for the technical design:

- Feedback on failing tests must be fast, such that the cause of the failure can be easily linked to the component changed.
- The location of the failing component must be reported as precisely as possible.
- The test description must be low in redundancy and be simple to adjust for changed behavior and new features.

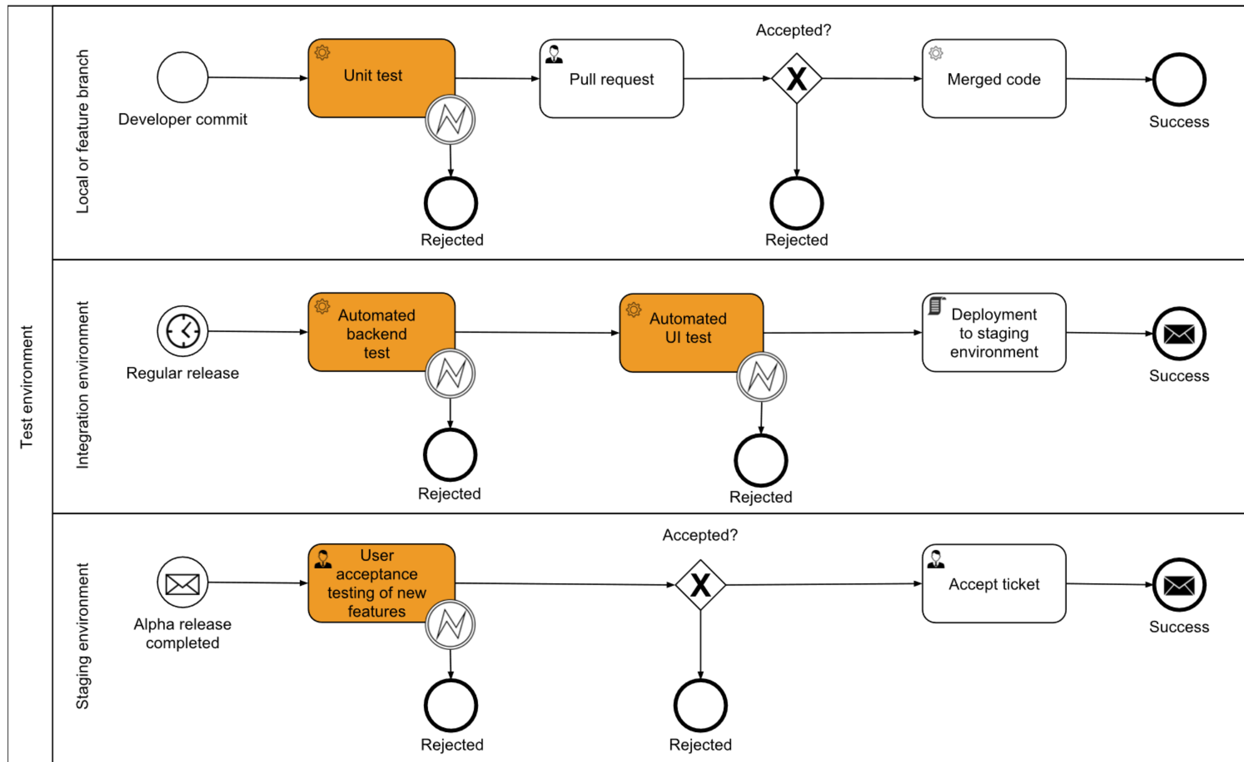
In order to achieve these goals, we had to accept our nature of failing humans. Whenever we did fail it was not enough to search and correct the error at hand. For every bug we fixed we also had to anticipate the next failure. We had to make it easier to find the error the next time and automate the procedures necessary for finding the bug.

Testing pipeline

Fast feedback on test results can only be achieved if tests start automatically at predefined triggers. We ran the following tests in an automated and regular fashion. They are ordered by fast and isolated to slow and integrated:

- **Unit tests**: Fast feedback, but simple and isolated test cases

Thetaris
Engineering in Finance



- **Backend tests**: Extensive coverage of functions Mostly regression testing, because server responses are hard to interpret
- **Automated UI tests**: Full coverage end to end, but fragile dependency on UI changes and often unclear location of responsibilities
- **Acceptance tests**: Full coverage, aesthetic requirements, fitness in real or close to real user scenarios

The tests were deployed in three different stages of deployment. The first stage ran a branched version of the source code, before it can be merged into the main branch. The second stage was automatically deployed, once a new feature was unit tested and merged. Two types of tests worked on the integration stage: Backend tests and automated ui tests. In the third stage everything is deployed in a near production setup. Everything that could not be tested automatically is left to this stage. Changes in design and user flows always land here.

Unit testing

Unit testing gives the fastest feedback on changing behavior of code. Unit tests must be written to run fast – few minutes at most – and be repeated with every commit. Various tools support this process. Bitbucket testing pipelines (see screenshot) can ensure that tests are run with every commit. The responsible committer and the changed code lines are directly linked to the error report. The effectiveness of unit tests can be monitored with test coverage tools. We used SonarQube and Istanbul to identify functionalities that were not sufficiently covered by unit tests. Manual commit reviews – pull requests – also helped us to ensure that test coverage stayed high.

Commits

All branches | Find commits

Author	Commit	Message		Date	Builds
	62081e5 M	1.19.4.3	release-1.19	an hour ago	
	3f9c59f	Fix version ...	release-1.19	an hour ago	
Stefan Dirnsto...	69e3852 M	1.19.4.2	release-1.19	3 hours ago	
	d111219	Fix tests again	release-1.19	3 hours ago	
Stefan Dirnsto...	9bbf5b5	added lodash	release-1.19	20 hours ago	
Stefan Dirnsto...	d7977ea	fix syntax	release-1.19	20 hours ago	
Stefan Dirnsto...	ce40fa4	fix syntax	release-1.19	20 hours ago	
	35d0eff	SUS-6911 fi...	release-1.19	20 hours ago	
	01b3b5d	SUS-6911: fi...	release-1.19	20 hours ago	

Useful service: test pipelines in Bitbucket. Every commit is labeled with the result status of the unit test.

Backend tests

The backends of our systems provide services through http restful requests. This is more or less the standard setup for any modern web enabled application. For simple requests it may be sufficient to test them with tools like postman or curl.

We found that thorough backend tests could only succeed with a more complex set of requests. We wanted to check the system in complex situations with sequences of requests that are not easy to create and verify manually.

We chose R markdown sheets to specify the tests. Markdown allowed us to create human readable test reports that contain all the code necessary to reproduce the error. We used visualizations to report the current behavior. This can be extremely useful for a human to analyze the nature of the error and save valuable debugging time.

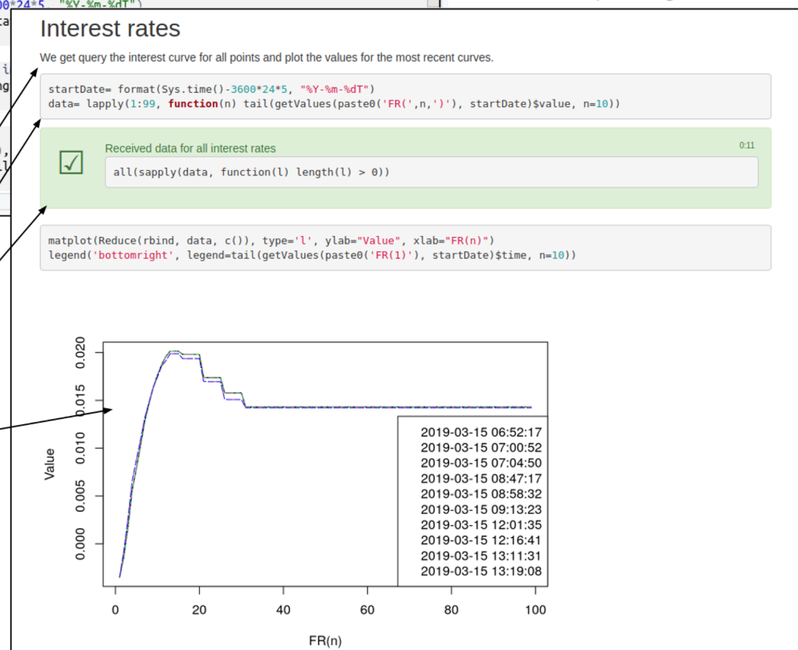
Markdown sheets offered the following benefits:

- A human readable test report is generated.
- All code is contained and easily accessible to reproduce the error.
- Visualization of test data helps to understand the current behavior and saves debugging time.
- Checks of the validity constraints are easy to define in a language specialized on data analysis.

UI Frontend tests

For UI test automation we had to train our testing software to interact with our application. Standard solutions like Selenium and Appium can be used to simulate the actual interaction, i.e. perform a click or a swipe on a selected element. However, we still had to specify the actual interaction sequence. There we had to face three challenges:

- Selecting the relevant regions of interaction is fragile. Buttons can change in layout, size and position depending on the device. They often change from version to version and tests need to keep up.
- Structuring the code for low redundancy is important. A generic change of one interaction pattern can be one line of code in the original source. Tests must be equally structured. Otherwise, tests become hard to maintain.
- Running the test on target devices must be automated. Cloud services provide simple interfaces for testing a large number of devices, operating systems and system setups.



Screenshot of a test case defined in markdown (background) and the rendered report (foreground).

Graphical element selection

We used two different location strategies for interaction elements on the screen. One was based on resource identifiers that are programmed into the source code. The other strategy is based purely on the visual representation. The graphical approach comes closest to how a human would interact with the software. For a test automation this requires the image detection to be sufficiently "smart" and cope with visual changes of shape, color or position of the interactive element.

We found that state of the art image recognition algorithms are very powerful. They can detect images across a range of variations in size and position. The major remaining task is the training. For this purpose, we wrote our own frontend that allowed us to label interaction elements. The image detection algorithm was sufficiently smart to detect the interaction element in different resolutions and layouts. Whenever a breaking change in layout occurred the element must be selected in the screenshot and relabeled.

We have developed our own web application for maintaining the training set of graphical assets. The design goals of the image classification frontend were:

- Fast and intuitive identification of relevant image elements
- Collection of relevant properties that allow the image to be identified in new layouts
- Precise reporting of failures
- Fast correction of image location strategies in case of breaking changes
- Low redundancy to minimize efforts after design changes

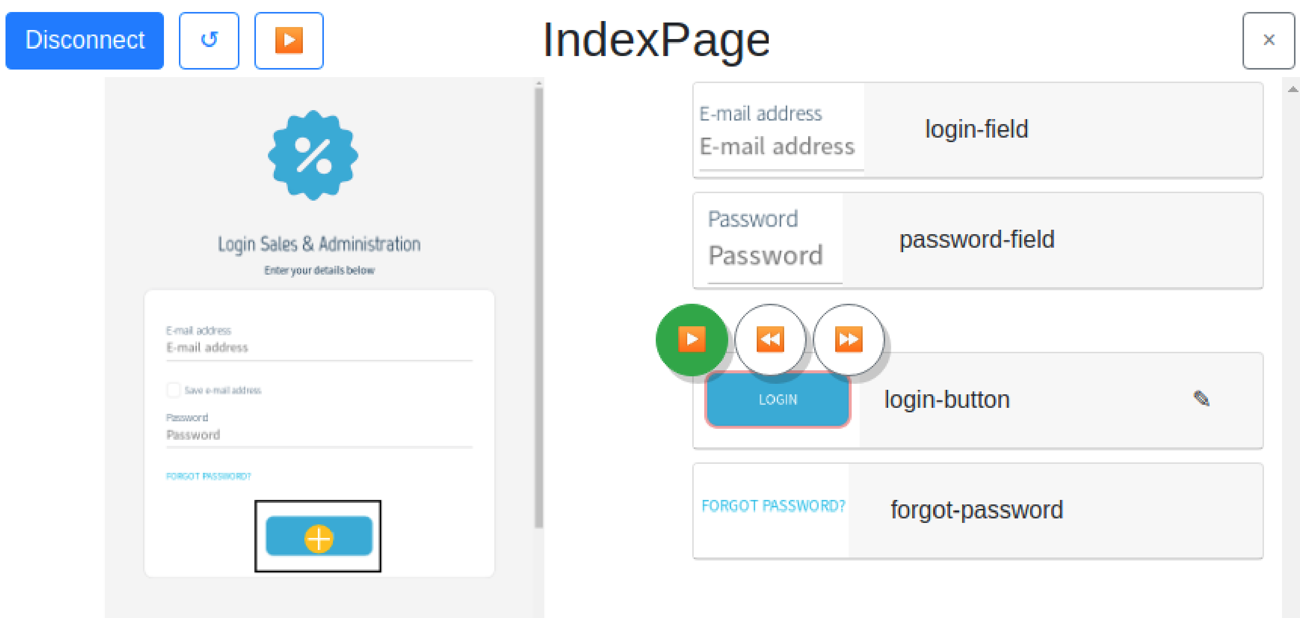
Page object pattern

The Page Object Model is the industry standard for test implementation. It leads to low redundancy in test code and is adopted throughout the industry. The approach works well with various element location strategies and keeps redundancies low.

We found that the graphical location strategy is preferable in situations where design is more or less stable and interaction with developers is costly. Element location-based on resource identifiers is preferable is situations where visual appearance of objects is highly dependent on context and developer time is available to add test hooks for relevant elements.

The page object pattern separates the code base into three parts:

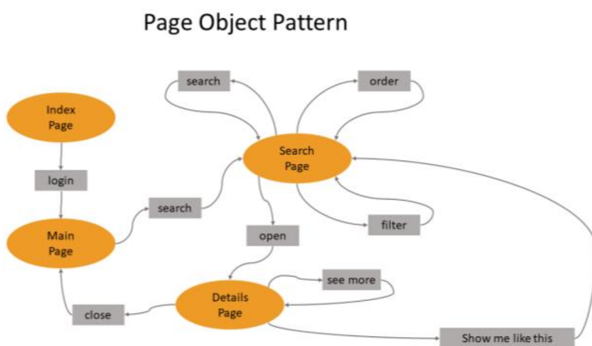
- Test steps,
  a programmatic description of steps as they would be performed by a user
- Page objects,
  a class hierarchy with methods for each transition between states. This part models the application states and defines the location strategy for interaction elements.
- Shared utilities,
  all algorithms required for reporting, access to the image classification data and image detection.



Screenshot of the image classification front end that we used to maintain the list of graphical assets. On the left you can see a screen shot of the application in the current state. On the right you see the list classified interaction elements

This design pattern has been found to lead effective test code with low redundancy. Changes in the application behavior can be covered with isolated changes in the page object model. A proper typed class structure of the page model also ensures that the test is consistent, i.e. that the operation performed in an application state is possible.

Example:
Assume a simple graph of application states and transitions below. For simplicity only a subsection of the application states and transitions are shown.



A simple code pattern the entry state "IndexPage" is shown in a simplified version below. The page object provides the method "login", which locates the interaction elements and returns a new state object named "MainPage". In this structure the code editor always knows what operations are possible in the current state. Refactoring tools of modern editors can be use efficiently.

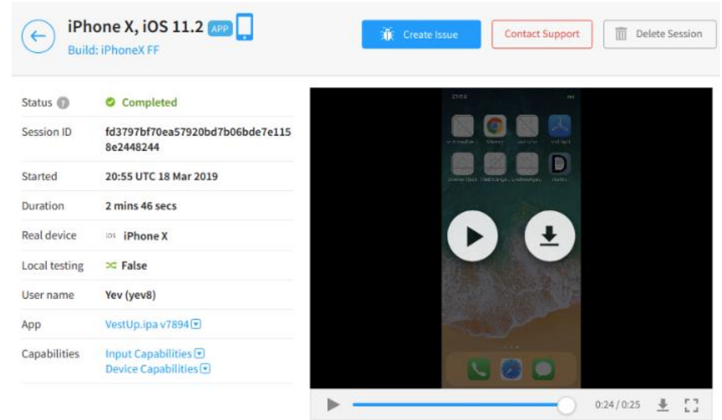
```
class IndexPage extends PageObject {
   MainPage login(String username, String password) {
       findElementByImage(getImage("login-field")).sendKeys(username)
       findElementByImage(getImage("password-field")).sendKeys(password)
       findElementByImage(getImage("login-button").click()
       return new MainPage();
   }
}
```

The page object model was first described in:

Improving test suites maintainability with the page object pattern: An industrial case study M Leotta, D Clerissi, F Ricca, C Spadaro - 2013 IEEE Sixth International Conference on Software 2013

Device testing

Running tests on large device numbers has become much less of a problem than it used to be. Cloud providers such as BrowserStack provide standard interfaces for Selenium and Appium based tests.



Screenshot of BrowserStack web interface. The properties of the selected device are shown next to a video of the application screen during the test.

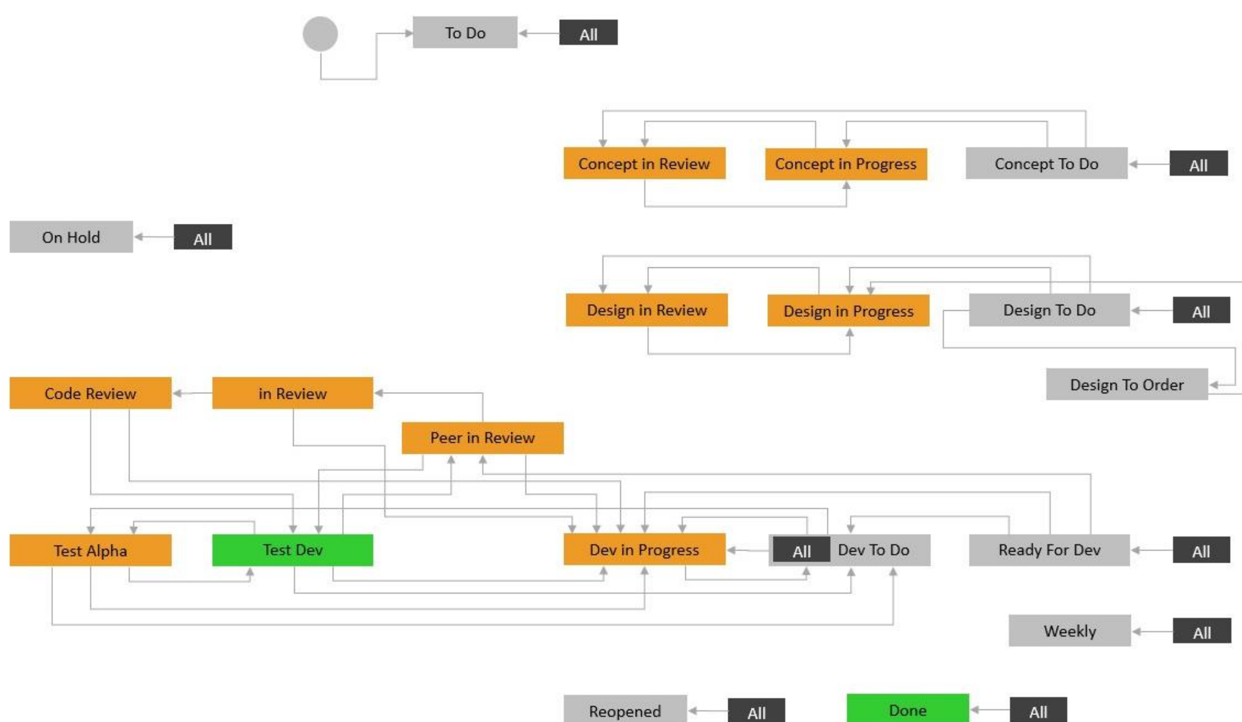
Acceptance tests

The final test in our pipeline was the acceptance test. This is where software developers finally present a new feature to the users or the business. This aspect is much more a human resource management task than it is technical.

We used the ticket management software JIRA to track the state of features and bugs. This formal tracking of states allowed us to keep all stake holders informed about the process. Additionally, we could measure our performance by the time from specification to acceptance or by the number of issues resolved.

We found the following states to be useful for successful test operation:

- Ready for development (Bug confirmed)
- Development in progress
- Peer review
  (manual code review, git pull request)



Screenshot ow the entire JIRA workflow containing the specification and design phase. Workflow transitions were associated with accepted or rejected paths.

- Testing in integration environment
  (TEST DEV = Unit tests passed, ready for integration testing)
- Testing on staging environment
  (TEST ALPHA = regression and integration tests passed, ready for acceptance testing)
- DONE
  Version number is supplied to the ticket

Conclusion

After some initial pain we could confirm: test automation significantly increased the stability of our software. We started with unit tests almost at the same time as we started coding. Later stages of test requirements were discovered successively as the complexity of our software grew. The backend tests were added first. They are relatively easy to set up and covered the entire functionality of the service. Frontend tests came later as interaction pattern stabilized. Designing frontend tests with a high level of maintainability was difficult and still is ongoing effort. The procedures around ticket management and test planning are also ongoing effort, because it essentially requires human effort to keep the system up-to-date.

This concludes this rather quick overview of our test setup. Some of the presented material is standard knowledge. Some of the presented tools were custom made. We hope that our insights can be useful for other projects. Don't hesitate to contact us for further details. We will be more than happy to help where we can.

Thetaris
Engineering in Finance

Please visit us:

www.thetaris.com

Or contact us at:

Leopoldstr. 244
D-80807 München

info@thetaris.com

+49 89 208 039 480