Supporting Agile Teams with a Test Analytics Platform: a Case Study

Olivier Liechti
University of Applied Sciences and Arts Western Switzerland
Yverdon-les-Bains, Switzerland
olivier.liechti@heig-vd.ch

Jacques Pasquier
Fribourg Univesity
Fribourg, Switzerland
jacques.pasquier@unifr.ch

Rodney Reis
Avalia Systems
Yverdon-les-Bains, Switzerland
rodney@avalia.systems

***Abstract*—Continuous improvement, feedback mechanisms and automated testing are cornerstones of agile methods. We introduce the concept of *test analytics*, which brings these three practices together. We illustrate the concept with an industrial case study and describe the experiments run by a team who had set a goal for itself to get better at testing. Beyond technical aspects, we explain how these experiments have changed the mindset and the behaviour of the team members. We then present an open source test analytics platform, later developed to share the positive learnings with the community. We describe the platform features and architecture and explain how it can be easily put to use. Before the conclusions, we explain how test analytics fits in the broader context of software analytics and present our ideas for future work.**

***Keywords*-agile development; automated testing; gamification; feedback channels;**

I. Introduction

Among key agile practices, three are at core of the case study presented in this paper. The first practice is *continuous self-improvement*: the repeated evaluation and adjustment of individual and collective practices in the pursuit of excellence and betterment of results. This is why agile teams conduct retrospective meetings, where they openly discuss strengths and weaknesses and decide how to experiment with new methods. The second practice is the implementation of *feedback cycles* to make progress and issues quickly visible to the team. Through these feedback cycles, people become aware of different aspects of software development such as code quality, process efficiency and team morale. The third practice is *automated testing*, which is essential to "bake quality in the product" while sustaining the pace of delivery and avoiding bottlenecks in the process.

These three practices relate to each other in different ways. One way to look at it is to consider that automated testing generates data, giving frequent feedback about the product quality. The team then uses this feedback to improve the code. Another way to look at it is to put the testing activity in the center. In this case, the team is made aware of events such as the creation and the execution of tests. People get feedback about the structure of the test suite (i.e. what types of tests cover what parts of the system) and its evolution over time. This helps the team continuously assess its testing practices and take corrective actions when needed. Ideally, the impact of automated testing on user satisfaction, velocity and team morale should be measured and made clearly visible. To capture these ideas, we propose the concept of *test analytics*. We define it as *analytics on test-related data in order to give actionable insights about product quality and agile practices, with the goal to support a continuous improvement process*.

Our understanding of *test analytics* has emerged from our experience at a startup company over several years. In the remaining sections of the paper, we start with a presentation of the case study. We explain how the implementation of test analytics has helped the team address different pain points, with a lasting impact. We then present an open source test analytics platform, which has later been developed to make our positive learnings available to other teams. We describe how the platform makes it easy to collect test-related data, to analyze it and finally to give feedback to the team. Before the conclusions, we explain how test analytics fits in the broader context of software analytics. We present some ideas for future work in this area.

II. Case study

When we started to experiment with the idea of test analytics, our goal was not to do scientific research. It was to help an agile development team get better and more consistent at testing, with the expectation of a broader impact: reduce the time needed to release new products, reduce the number of customer issues, and increase employee pride and satisfaction. In the following paragraphs, we first set the context by introducing the company and the team. We then describe the pain points felt by the team and analyze their causes. We then explain our attempts to trigger both perspective and behaviour changes within the company. Finally, we present our results and the key lessons learned through the process.

A. Context

In the Swiss higher education system, universities of applied sciences have a strong practical focus. A lot of the research projects are done in collaboration with commercial companies, with the goal to bring innovative products and services to the market. In early 2008, the first author started a collaboration with an early stage start-up, which had set to build a global mobile payment platform. The two founders had a business background. They had a prototype and a first customer, but no technical team. Initially, the research institute received the mandate to transform the prototype into an industrial-grade platform. Over time, the first author was asked to take increasing responsibilities and eventually became CTO for the company. He grew the engineering team, many members of which were former students and research engineers from the institute. The team was young and eager to learn. The team culture was imprinted with agile values: autonomy, trust, responsibility, and continuous learning. The work environment was typical of a startup company. Everything was changing rapidly and often, including business opportunities, product priorities, and customer requirements. There was a sense of urgency and deadlines were often agressive. Over a few years, the team grew to reach about 25 engineers distributed across Switzerland, Romania and Singapore.

At a certain stage, the team started to feel the pain of the technical and organizational debt that had been accumulated over time. The most severe consequence was the increasing time required to release new versions of the software. There were far too many back and forths between the software and QA engineers, and deployments were stressful. The team was working on two-week iterations, but the definition of *done* was not crisp nor strictly adhered to. This negatively impacted team satisfaction and confidence. On the bright side, the environment was healthy from an agile perspective, in the sense that it was possible to openly put the issues on the table and there was a willingness to improve things. The self-diagnosis by the team made it clear that both engineering practices and collaboration practices had to be considered to improve the situation. In other words, the team had to pay back technical and organizational debt.

B. Technical debt and the chore of writing tests

While the software architecture and the code were not perfect, they were not the real problem. The foundations were good and the code had been refactored on a regular basis. The main problem was the lack of a proper test harness. There were *some* unit tests, there were *some* integration tests and some very interesting tools had even been developed in-house to test mobile components. However, there was no discipline around testing. Test-driven and behaviour-driven development were not part of the standard workflow. Most engineers agreed with the theory of extreme programming, but in practice viewed development and automated testing as two distinct activities. Writing automated tests was not seen as something intellectually rewarding, but rather as a chore. In this context, it was easy for the team to use time pressure as a reason to postpone these activities and to rather dive into the development of new features. This attitude is not specific to this particular team and is something we have observed over and over in other contexts.

To address this problem, it was necessary to trigger a behaviour change and then to measure the impact of a disciplined testing process in tangible ways. We believed that this would be key to increase personal satisfaction and to intrinsically motivate the team in a sustainable way.

C. Organizational debt and the whole-team approach

Agile methods put an emphasis on collaboration, autonomy and collective responsibility. When the development team is entrusted with some objectives, it should be able to decide how to organize itself to achieve them. Everybody in the team should share this responsibility and proactively contribute to the team effort. In the early days, the notion of team was very developer-centric. Typically, a development team would use agile practices to deliver *deployable* software on a regular basis. This software would be handed over to a QA team with the responsibility to fully test and validate it. Finally, the validated software would be handed over to an IT Operations team with the responsibility to deploy and operate it. The software, QA, and IT Ops engineers would often belong do different organizations. Today, the flaws of this type of setup are widely recognized and many organizations are moving towards the *whole-team* approach. The idea is that the team responsible for a product or a set of features should be cross-functional. There should be no handover between organizations and the same small team should assume the responsibility for the entire product lifecycle. This idea is central to *DevOps* [1] and *agile testing* [2].

However, the organizational structure at the company had followed the traditional path: there were distinct development, QA, and IT Ops teams. The company was small, but we still had handovers between teams. The negative consequences that we experienced were fully consistent with the theory. As mentioned before, most developers were not naturally inclined to put effort in automated testing. It was too easy for them to assume that anything related to testing was the responsibility of the QA staff. Since the QA staff was doing mostly manual testing, they were unable to keep up with the pace of development. Their choice was to either test a subset of the functionalities developed or to postpone the release dates.

To address this problem, it was necessary to shake up the organization and to move towards a true *whole-team* setup. The boundaries between the roles of software and QA engineers had to blur. Developers needed to accept automated testing as their primary responsibility and truly apply extreme programming practices. The QA engineers would then have time to do more exploratory testing and bring more value to the product. To help the transition, we believed that the introduction of behaviour-driven development was promising.

D. Driving behaviour change

With a shared agreement on the problems and their causes within the team, it was then time to experiment with new practices and to see whether we could trigger and sustain positive behaviour changes. Interestingly enough, one of the products developed by the company had a comparable objective. As part of a mobile commerce platform, this product intended to drive the consumer behaviour (e.g. increase the number of in-app purchases). This product was an example of *persuasive technology* [3] and we had become familiar with the Fogg Behaviour Model (FBM) [4]. The FBM claims that to perform

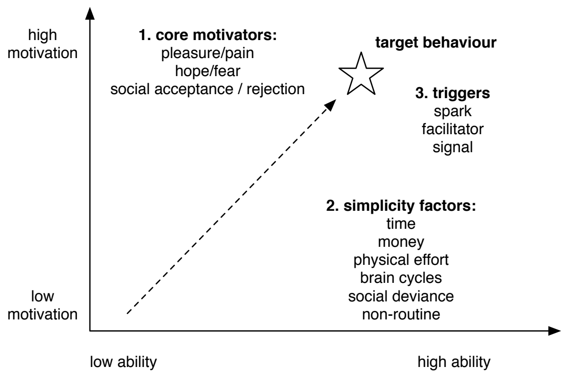


Fig. 1. The Fogg Behaviour Model [4]

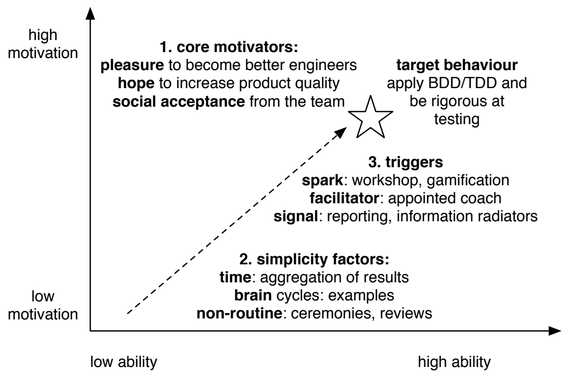

high motivation
1. core motivators:
pleasure to become better engineers
hope to increase product quality
social acceptance from the team
target behaviour
apply BDD/TDD and
be rigorous at
testing
3. triggers
spark: workshop, gamification
facilitator: appointed coach
signal: reporting, information radiators
2. simplicity factors:
time: aggregation of results
brain cycles: examples
non-routine: ceremonies, reviews
low motivation
low ability
high ability

Fig. 2. The Fogg Behaviour Model applied to agile testing

a target behaviour, a person must i) be sufficiently motivated, ii) have the ability to perform it, and iii) be triggered to perform it. A visual representation of the FBM is shown in Figure 1, with its three factors: *motivators*, *simplicity factors* and *triggers*. They all need to be considered in order to drive a person to adopt a target behaviour and continue performing it. Triggers *tell people to perform a behaviour now*. Fogg describes three types of triggers. *Sparks* are used when people lack motivation and *facilitators* are used when they lack abilities. *Signals* are used when they have the motivation and ability to perform the behaviour simply acting as a reminder.

E. Experiments

To address the technical and organizational problems described before, we experimented with various ideas and iteratively refined them. Eventually, the most useful practices became habits for the team.

We started with a focus on the collaborative practices, with the goal to move towards the *whole-team approach*. A key element of Behaviour-Driven Development (BDD) [5] is the notion of *ubiquitous language*, borrowed from domain-driven design [6]. The ubiquitous language helps the different stakeholders (developers, business analysts, testers, etc.) to communicate about the product. It can also be used to write *executable specifications*, which capture the externally visible behaviour of the product. Our hypothesis was that introducing BDD in our workflow would serve two purposes. Firstly, it would help us increase the collaboration between people with different roles in the company. We decided that executable specifications would need to be written, or at least reviewed, by four people: a product owner, a QA specialist, an IT ops engineer and a developer. Secondly, we also decided that the executable specifications would serve as the primary artefact for accepting a feature as *done*. At the end of the development workflow, the multi-disciplinary feature team would present and execute the specification to introduce the new functionality to the other teams. To implement the transition to BDD, we organized a series of workshops to train the team, started with small experiments on simple features and appointed people to become experts and coaches. These small experiments have allowed us to evaluate and select the tools, and to create a collection of examples, which made it easier for the rest of the team to get started.

In terms of engineering practices, our goal was to drive the software engineers to be more thorough with automated testing. The overall company focus on BDD helped, because it provided a context and overarching goal. However, we also wanted to improve other areas of testing, such as unit testing and non-functional testing. We also had some ideas to use gamification to add some fun to the process. We have started by designing a simple system that collected the results produced by test runners, such as JUnit, in a central location. This has allowed us to detect the addition of new tests to the test suite. For every test, we knew the category (unit, integration, etc.), the author, the tested component and the underlying language. On the fun side, we were then be able to award points and badges to good contributors and generate testing leaderboards. On the serious side, we were able to generate a single report that aggregated test results across components, layers and languages. The report not only presented a snapshot of the product quality, but also its evolution. We started the implementation of this system in the context of the annual company hackathon, which had been initiated a couple of years before to promote creativity and team building. The first version was rapidly put together and later on, some effort was put to extend it iteratively.

F. Key learnings

The experiments had a significant and lasting impact on the team. The team was eager to improve, which obviously made things a lot easier. The adoption of BDD by the whole team was fast and effective in changing the mindsets. One thing that the team had to fine tune was the granularity of the executable specifications. Initially, every feature was fully specified by a team of four people. It was a good thing because it got the ball rolling and helped us get alignment across roles. After a while, the team felt that for some of the features, the overhead was too big (people had the feeling that they spent too much time in meetings). The process was adjusted, so that only one or two people would write the specification, which would then be reviewed by the others. A successful review of the specification would then make it ready for development.

The gamification features played their role in creating some initial excitement. As developers acquired testing skills and started to see a tangible impact on quality, they had enough intrinsic motivation to sustain the effort. At this stage, the user interface that presented the aggregated test results and the evolution of the test suite became the preferred feedback channel for the team. Monitoring the characteristics of the test suite (by category, component, language) allowed the team to monitor progress and make adjustments. Getting a regular feedback across layers and up to the externally visible behaviour tremendously increased the team's confidence, pride, and satisfaction. Several years after these experiments, the engineers have moved on with their careers and are working in other companies. Many of them have reported that they have brought the acquired testing culture with them and continue to use some of the practices. Some have faced resistance and have expressed frustration not being allowed to invest the proper time in automated testing.

In Figure 2, we use the FBM as a framework to summarize our experiments. Our goal was to drive two target behaviours by the team: i) the adoption of BDD by all stakeholders and ii) the rigorous maintenance of an automated test suite. We leveraged different factors to increase motivation: the pleasure to master new skills and to become better professionals, the hope to have an impact on quality and time to release and the appreciation of the other team members. We relied on other factors to reduce friction: the ability to easily get aggregated test results saved time, a comprehensive collection of examples reduced cognitive overhead and test-related ceremonies introduced new routines. Finally, we used different triggers to set the team in motion. The workshops allowed the team to reflect and understand the broader impact that was expected. The gamification features gave the initial incentive to write tests. The appointment of testing coaches helped the whole team acquire the required skills. Finally, notification of test results via different channels was a continuous reminder of the testing activity and of the progress made by the team.

III. An Open source test analytics platform

We believe that the positive results that we observed are not specific to that particular team. This has led us to start an open source project, with the goal to make some of the learnings available to other teams. Today, this platform is publicly available [7] and can be setup in a variety of technical environments. In the following paragraphs, we first describe the architecture and the execution model of the platform. We then present the characteristics of its core components and describe three types of feedback channels that it enables.

A. Architecture and execution model

The architecture of the test analytics platform is shown in Figure 3. At the lowest layer, *probes* capture test results produced by different tools (unit test frameworks, BDD frameworks, performance testing tools, etc.). Probes can be installed wherever tests are run, i.e. both on developers machines and build servers. In the middle layer, the server exposes interfaces

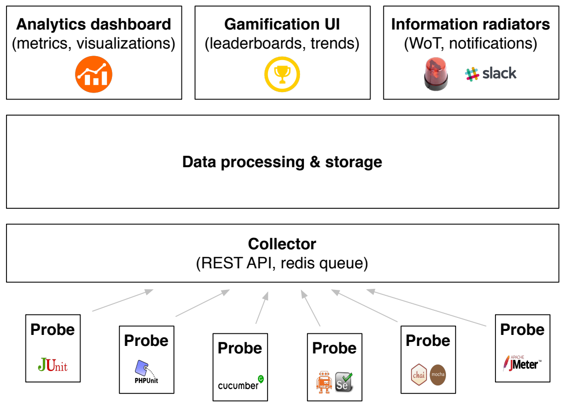


Fig. 3. Architecture of the test analytics platform

to collect the data sent by the probes. It processes the data and keeps the entire history in persistent storage. In the upper layer, different feedback channels make the information and the derived insights available to the team.

The execution model is shown in Figure 4. It is important to understand the structure of the payloads generated by the probes, as well as the capabilities of the platform. The model is based on continuous delivery [8] concepts:

- *Pipelines*. A pipeline defines a sequence of stages required to build and validate the software. Most development teams define several pipelines. Developers may use a simple pipeline on their machine, with one phase to compile the code and one phase to run unit and integration tests. The continuous integration server may use a longer pipeline with phases dedicated to automated User Acceptance Testing (UAT) and performance testing. The same pipeline can be executed in different environments (on a developer's machine, on a CI server, etc.).
- *Stages*. Within a pipeline, a stage defines a logical group of operations (steps). In a typical pipeline, there are a few stages such as the commit stage, the integration testing stage, the UAT stage. It is up to the development team to define the granularity and the meaning of stages.
- *Steps*. A step is an operation that is executed during a stage of pipeline. Compiling source files and running unit tests are two examples of steps.
- *Nodes*. A node is a computer on which a step is executed. For instance, it can be a developer machine, the integration server or one of the build servers in a build cluster. A node has certain properties: operating system, available resources, etc.
- *Pipeline executions*. The execution of a pipeline starts whenever the software has to be built and validated. Sometimes, developers trigger the execution of a newe (in the IDE, in the CI UI, on the command line). Sometimes, the execution is triggered as a side effect (e.g. after a commit).
- *Test execution and payload generation*. During a pipeline execution, one or more steps may cause a probe to send a test payload to the server. For example, in the commit

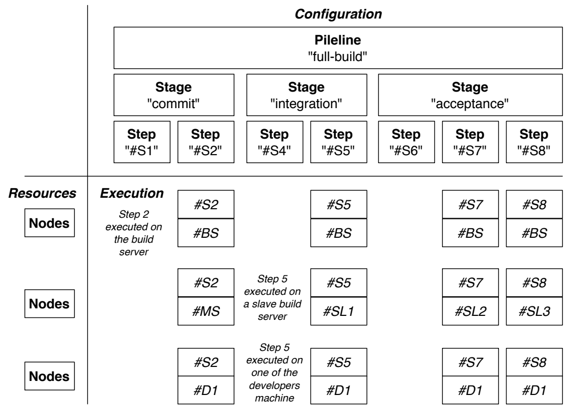


Fig. 4. Pipelines, nodes and test executions

stage of the developer pipeline, a step might execute a suite of unit tests via maven, which would cause the JUnit probe to send a payload. This would happen on a single node. Another example would be, in the integration stage of the daily build pipeline, a step that executes a suite of API tests and another step that executes a suite of integration tests. Two probes would generate two payloads and send them to the server. This may happen on the same node or on different nodes. *A test payload is therefore associated with a pipeline execution, a stage, a step and a node*.

B. Probes

The test analytics platform is not a substitute to test frameworks and tools. The role of the platform is to collect the results produced by the existing test infrastructure and to transform this raw data in actionable information. The integration between the platform and the third-party frameworks is done via probes. Simply stated, a probe is a framework-specific agent that is able to collect the results of every test run and to forward it to the platform in a normalized format.

At the time of writing, the probes for the following test frameworks are already available: JUnit, Arquillian, Jasmine, Mocha, Karma, PHPUnit, Python nose2, pytest, Cucumber, RSpec. These probes are implemented on top of shared libraries for the following languages: Java, JavaScript, Python and Ruby. The librairies make it easy to implement new probes for other test frameworks. Essentially, it is sufficient to be notified when new test results are available (usually via a listener API provided by the test framework) and to pass them to the language-specific library. The library then generates and uploads a JSON payload. The structure of this payload is shown in Listings 1 and 2.

C. Feedback channels

Different types of channels can be combined to present information and insights to the team. The first one is the *analytics dashboard*, shown in Figures 5 and 6. This is a *pull* channel, in the sense that users must take the conscious

```
{
  "version": 1,
  "project": {},
  "scm": {},
  "pipeline": {},
  "node": {},
  "runtime": {},
  "testFramework": {},
  "results": []
}
```

Listing 1. Structure of the JSON payload sent by probes

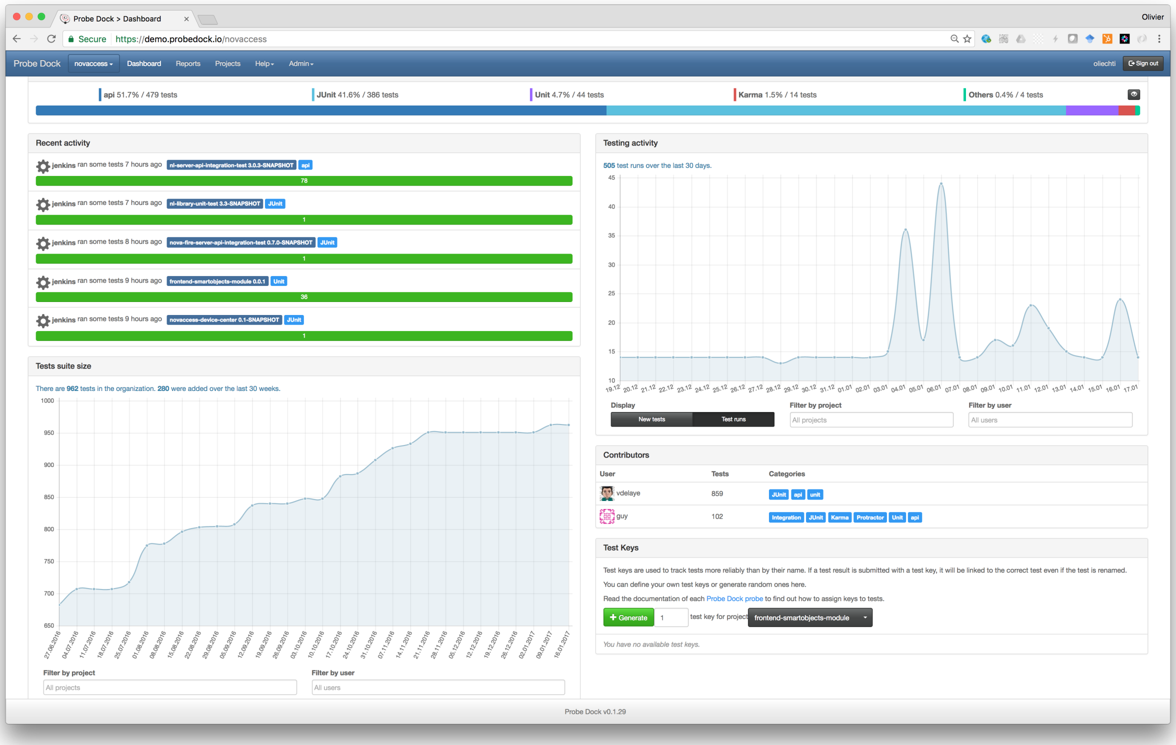
```
{
  "key": "8srng9a",
  "name": "Login: user should be able to log in",
  "passed": false,
  "duration": 1204.35,
  "tags": [ "security" ],
  "tickets": [ "JIRA-190", "JIRA-8320" ],
  "log": "2016-03-02T14:49:34.781Z [DEBUG] - Opened http://localhost:3000\n2016
        -03-02T14:49:34.781Z [DEBUG] - Clicked on #login",
  "failures": [
    { "type": "Failed Assertion", "message": "Expected loggedIn to be true, got
          false", "stackTrace": "foo\nbar" },
    { "type": "Error", "stackTrace": "foo\nbar\nbaz" }
  ],
  "file": "src/test/resources/com/example/users/LoginFlow.java",
  "line": 129,
  "hierarchy": [ "com", "example", "users", "LoginFlow" ],
  "runtime": {
    "package": "com.example.users",
    "class": "Login",
    "method": "userShouldBeAbleToLogIn"
  },
  "memory": 12638380
}
```

Listing 2. Example of a test result in the payload

decision to visit the dashboard and to analyze the situation. Here are some of the features supported by the interface:

- in the upper-left corner of Figure 5, the recent activity shows which tests have been run, in which environments and what have been the results. This shows what is the current status of the build. It also indicates what individual developers are currently working on.
- In the lower-left corner of Figure 5, the widget shows the evolution of the test suite over time. By default, the accumulated number of all tests is shown, but it is possible to filter the test suite by project or developer. A flat line indicates that no new test has been added for some time, which suggests an increasing technical debt.
- In the upper right corner of Figure 5, the number of test runs is shown in a line graph. Users can also use different filters to analyze the activity in more details.
- In the upper part of Figure 6, users get an overview of a specific test run. The visual indicators show the relative and absolute number of failing tests.
- For every test, users can dig into the details and inspect the metadata. As shown in the lower part of Figure 6, they even have access to the execution trace (this information is sent by the probes together with the test results).

The second feedback channel is the user interface that embodies the *gamification* features. Based on the lessons we learned during the case study, we initially invested less time on gamification than on analytics. Today, the stable release of the test platform provides basic mechanisms for team members to evaluate their contributions compared to their peers. However, the metrics are not presented with game-specific terminology (e.g. badges, reputation, etc.). We have plans to change that



api 51.7% / 479 tests | **JUnit** 41.6% / 386 tests | **Unit** 4.7% / 44 tests | **Karma** 1.5% / 14 tests | **Others** 0.4% / 4 tests

Recent activity

jenkins ran some tests 7 hours ago nl-server-api-integration-test 3.0.3-SNAPSHOT api — 78

jenkins ran some tests 7 hours ago nl-library-unit-test 3.3-SNAPSHOT JUnit — 1

jenkins ran some tests 8 hours ago nova-fire-server-api-integration-test 0.7.0-SNAPSHOT JUnit — 1

jenkins ran some tests 9 hours ago frontend-smartobjects-module 0.0.1 Unit — 36

jenkins ran some tests 9 hours ago novaccess-device-center 0.1-SNAPSHOT JUnit — 1

Tests suite size

There are 962 tests in the organization. 280 were added over the last 30 weeks.

Filter by project: All projects — Filter by user: All users

Testing activity

505 test runs over the last 30 days.

Display: New tests | Test runs — Filter by project: All projects — Filter by user: All users

Contributors

User	Tests	Categories
vdelaye	859	JUnit api unit
guy	102	Integration JUnit Karma Protractor Unit api

Test Keys

Test keys are used to track tests more reliably than by their name. If a test result is submitted with a test key, it will be linked to the correct test even if the test is renamed.

You can define your own test keys or generate random ones here.

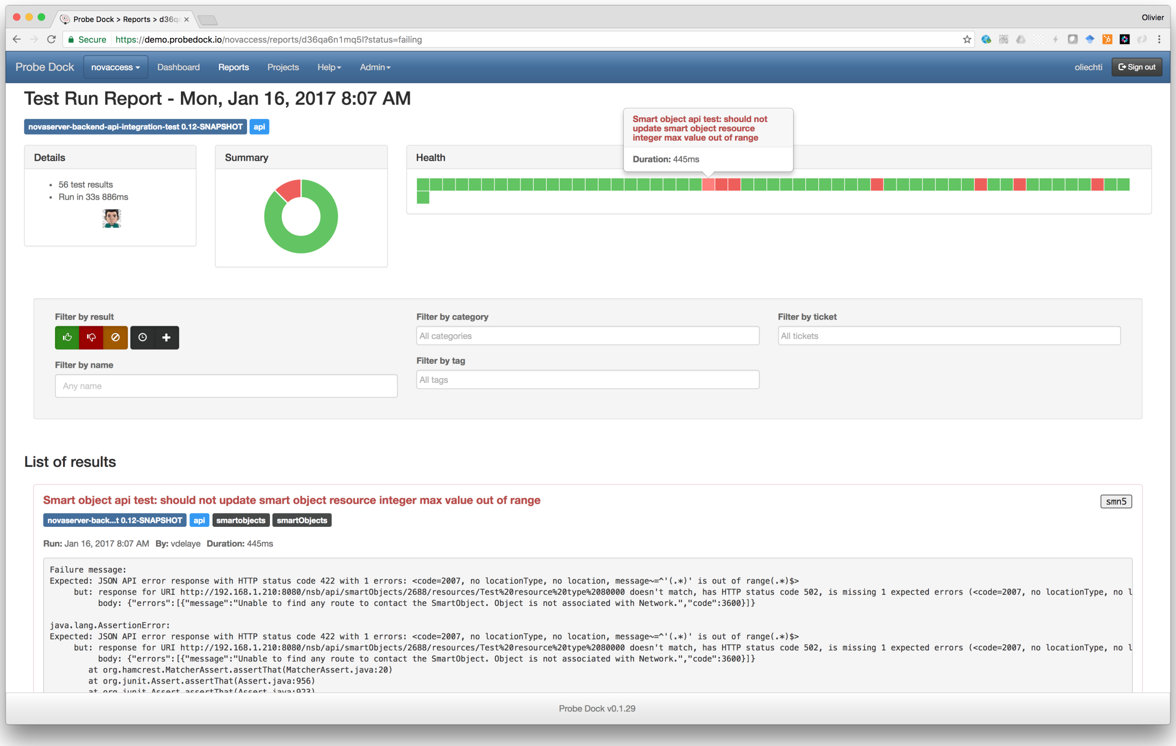
Read the documentation of each Probe Dock probe to find out how to assign keys to tests.

+ Generate 1 test key for project frontend-smartobjects-module

You have no available test keys.

Probe Dock v0.1.29

Fig. 5. Overview of test-related activity in the dashboard


Probe Dock > Reports > d36q
https://demo.probedock.io/novaccess/reports/d36qa6n1mq5l?status=failing
Probe Dock
novaccess
Dashboard
Reports
Projects
Help
Admin
oliechti
Sign out

Test Run Report - Mon, Jan 16, 2017 8:07 AM

novaserver-backend-api-integration-test 0.12-SNAPSHOT api

Details

- 56 test results
- Run in 33s 886ms

Summary

Health

Smart object api test: should not update smart object resource integer max value out of range

Duration: 445ms

Filter by result — Filter by category: All categories — Filter by ticket: All tickets

Filter by name: Any name — Filter by tag: All tags

List of results

Smart object api test: should not update smart object resource integer max value out of range smn5

novaserver-back...t 0.12-SNAPSHOT api smartobjects smartObjects

Run: Jan 16, 2017 8:07 AM **By:** vdelaye **Duration:** 445ms

```
Failure message:
Expected: JSON API error response with HTTP status code 422 with 1 errors: <code=2007, no locationType, no location, message~=^'(.*)' is out of range(.*)$>
     but: response for URI http://192.168.1.210:8080/nsb/api/smartObjects/2688/resources/Test%20resource%20type%2080000 doesn't match, has HTTP status code 502, is missing 1 expected errors (<code=2007, no locationType, no l
          body: {"errors":[{"message":"Unable to find any route to contact the SmartObject. Object is not associated with Network.","code":3600}]}

java.lang.AssertionError:
Expected: JSON API error response with HTTP status code 422 with 1 errors: <code=2007, no locationType, no location, message~=^'(.*)' is out of range(.*)$>
     but: response for URI http://192.168.1.210:8080/nsb/api/smartObjects/2688/resources/Test%20resource%20type%2080000 doesn't match, has HTTP status code 502, is missing 1 expected errors (<code=2007, no locationType, no l
          body: {"errors":[{"message":"Unable to find any route to contact the SmartObject. Object is not associated with Network.","code":3600}]}
        at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)
        at org.junit.Assert.assertThat(Assert.java:956)
```

Probe Dock v0.1.29

Fig. 6. Drill-down into the details of a test execution

in the future. We have already developed a generic gamification platform [9] as a new loosely coupled component in the architecture. The multi-tenant platform allows third-party applications to send streams of events that capture user activity. Rules can be defined with a Domain Specific Language (DSL) to react to user events and to trigger game mechanisms (rewards, progression, etc.). An API is provided to retrieve the state of players and display it. To drive the specification of the gamification platform, we used the gamification of agile processes as the first application domain. Every activity performed by a team member (writing code, writing tests, reviewing code, breaking a build, etc.) is an event that can go through the rule-based system. It is also worth noting that when we developed the gamification platform, we used the test analytics platform to monitor its quality. The lead developer, who did not have a lot of experience with automated testing also became "test addicted".

The third feedback channel serves a very different purpose and is a *push* channel. In this case, the goal is to notify users about the ongoing activity without requiring the focus of their attention. This is best done with user interfaces that are physically embedded in the environment and that use relatively abstract representations. The goal of these representations, is to allow users to continuously and effortlessly maintain awareness about what is happening. They are also an example of signal triggers in the FBM, by reminding people to perform the target behaviour. Similarly to the gamification features, we are working on this topic from a broader perspective and consider other types developer activities. We have developed a series of prototypes to explore how the activity of a development team could be represented with connected objects [10]. The impact of awareness on motivation is discussed in [11].

IV. Future work and conclusions

Our definition for *test analytics* is based on the broader definition for *software analytics* [12]. Software data can be extracted from all sorts of repositories: version management systems, issue trackers, mailing lists, etc. Mining software repositories sees growing interest from the research community. For instance, there have been attempts to model the evolution of software systems and ecosystems, to predict defects and to understand the effectiveness of development processes by applying data mining, machine learning and visualization techniques. Some of the work has specifically looked at automated testing, for example to study the co-evolution of application and test code [13], as well as the traceability between code units and automated tests [14].

In that spirit, our goal is now to analyze testing practices across a large number of teams and projects. The resulting data will allow agile teams to compare their practices against their peers. Beyond simple metrics, we aim to measure the impact of automated testing on defects, frequency of releases, developer satisfaction and other dimensions. This would be of tremendous value, as it would provide motivation and guidance to the teams, as well as a way to measure their progress. As an initial step towards this ambitious goal, we have created a flexible and scalable data collection platform. We will now use it in concert with the test analytics platform. Our first goal is, for a given project, to retrieve the list of all past releases, to run the corresponding test suite and to collect the results. Our second goal is to perform this task on a large number of curated open source projects. The other metrics, such as the number of issues, used to measure the impact still need to be defined.

It is hard to dispute the fact that automated testing significantly improves software quality. In many environments, however, there is room for improvement. In our experience, there are simple and effective ways to make the impact and the progress visible to the team with a huge boot on motivation. A test analytics platform offers a way to get started.

Acknowledgment

The authors would like to thank the entire team, whose efforts have been reported in the case study. The learnings are the results of a collective and participative effort. The authors would also like to thank the contributors to the open source platforms described in the paper, and in particular the research engineers and students at the University of Applied Sciences and Arts Western Switzerland and Fribourg University.

References

[1] M. Walls, *Building a DevOps culture*. O'Reilly Media, Inc., 2013.

[2] L. Crispin and J. Gregory, *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.

[3] B. J. Fogg, "Persuasive technology: Using computers to change what we think and do," *Ubiquity*, vol. 2002, no. December, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/764008.763957

[4] B. Fogg, "A behavior model for persuasive design," in *Proceedings of the 4th International Conference on Persuasive Technology*, ser. Persuasive '09. New York, NY, USA: ACM, 2009, pp. 40:1–40:7. [Online]. Available: http://doi.acm.org/10.1145/1541948.1541999

[5] C. Solis and X. Wang, "A study of the characteristics of behaviour driven development," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2011, pp. 383–387.

[6] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[7] (2017, January). [Online]. Available: http://probedock.io

[8] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[9] Y. Kammoun, "Game dock: design and implementation of a gamification platform," University of Applied Sciences Western Switzerland, Bachelor thesis, 2016.

[10] O. Liechti, J. Pasquier, L. Prévost, and P. Gremaud, "The wot as an awareness booster in agile development workspaces," in *International Conference on Web Engineering*. Springer, 2016, pp. 598–602.

[11] O. Liechti, J. Pasquier, and R. Reis, "Beyond dashboards: on the many facets of metrics and feedback in agile organizations," in *10th International Workshop on Cooperative and Human Aspects of Software Engineering (submitted)*, 2017.

[12] T. Menzies and T. Zimmermann, "Software analytics: so what?" *IEEE Software*, vol. 30, no. 4, pp. 31–37, 2013.

[13] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *2008 1st International Conference on Software Testing, Verification, and Validation*. IEEE, 2008, pp. 220–229.

[14] B. Van Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE, 2009, pp. 209–218.