Achieving Test Automation with Testers without Coding Skills: An Industrial Report

Davrondzhon Gafurov
Arne Erik Hurum
Martin Markman
Norwegian Directorate of eHealth
Oslo, Norway
davrondzhon.gafurov@ehelse.no

ABSTRACT

We present a process driven test automation solution which enables delegating (part of) automation tasks from test automation engineer (expensive resource) to test analyst (non-developer, less expensive). In our approach, a test automation engineer implements test steps (or actions) which are executed automatically. Such automated test steps represent user actions in the system under test and specified by a natural language which is understandable by a non-technical person. Then, a test analyst with a domain knowledge organizes automated steps combined with test input to create an automated test case. It should be emphasized that the test analyst does not need to possess programming skills to create, modify or execute automated test cases. We refine benchmark test automation architecture to be better suitable for an effective separation and sharing of responsibilities between the test automation engineer (with coding skills) and test analyst (with a domain knowledge). In addition, we propose a metric to empirically estimate cooperation between test automation engineer and test analyst's works. The proposed automation solution has been defined based on our experience in the development and maintenance of Helsenorge, the national electronic health services in Norway which has had over one million of visits per month past year, and we still use it to automate the execution of regression tests.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Test automation, process-driven test automation, keyword-driven test automation, DSL for test automation, Helsenorge

ACM Reference Format:
Davrondzhon Gafurov, Arne Erik Hurum, and Martin Markman. 2018. Achieving Test Automation with Testers without Coding Skills: An Industrial Report. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3238147.3240463

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240463

1 INTRODUCTION

Time to market, quicker feedbacks from end-users, risk of large changes and similar factors push companies and organizations towards continuous or more often software product releases. Testing is an essential part of any software development life cycle model [9]. Consequently, such frequent deliveries have direct impact on test activity. This is especially true for regression testing due to its natural growth because today's testing of a new feature becomes tomorrow's regression test. To cope with increasing demand on frequent testing, usually with limited resources, test automation is an important strategy to address this challenge. However, from one hand, a test automation is neither simple nor straightforward [4]. In addition to the process adjustment, it requires a set of high technical skills, important of which is the ability of writing a computer code. On the other hand, usually a test automation engineer who possesses high programming skills with a testing-mindset is scarcer, when compared to a typical test analyst (functional tester) who does not need to have programming experience or a technical background.

A test step is an action or process to be performed on the system under test. It is typically equivalent to the action of an end-user while using the system. Examples of a test step can be "signing in to the system", "signing out from the system", "agreeing to the terms of use", "disagreeing to the terms of use", "sending a message", "deleting a message" and so on. A test case is a sequence of logically organized test steps that verifies desired system functionality. We refer to the test case which can be executed automatically as an automated test case (ATC). A test automation can be implemented by applying different techniques, ranging from a simple capture and replay and finishing with more sophisticated ones such as keyword- and process- driven approaches [1, 12]. Another term which is used for process driven test automation is domain specific languages (DSL) for test automation, for instance [8].

In this paper, we present a process driven test automation solution which allows test cases to be defined from a workflow perspective [1]. In our approach, a test automation engineer implements test steps which are executed automatically. Such automated test steps represent user actions on the system under test and specified by a natural language which is understandable by a non-technical person with a domain knowledge. Then, a test analyst organizes

automated steps combined with test input to create an automated test case. It should be emphasized that the test analyst does not need to possess programming skills to create, update or execute automated test cases. We refine benchmark test automation architecture to be better suitable for effective separation and sharing of responsibilities between a test automation engineer (with coding skills) and a test analyst (with domain knowledge). In addition, we propose a metric to empirically estimate impact of test automation engineer's work on the work of test analyst. We implement and apply our test automation concept on real-world web application, namely automating regression test of national e-health Internet portal in Norway – Helsenorge [5].

The main contribution of the paper is on leveraging from our experience with Helsenorge and developing a test automation solution to move (part of) automation work from test automation engineer (developer, expensive resource) to a test analyst (non-developer, less expensive resource). Our approach also contributes towards a better cross-functional software development teams, since a test analyst in the development team can carry out test automation tasks without depending on others than team members.

The rest of the paper is structured as follow. Section 2 briefly presents an overview of Helsenorge and Section 3 describes our test automation solution. Section 4 presents a metric for test creation efficiency. Section 5 summarizes main benefits, limitations and lessons learned, and section 6 concludes the paper.

2 SYSTEM UNDER TEST – HELSENORGE

2.1 Helsenorge Overview

Our system under test (SUT) is Helsenorge. It is a national portal of e-health services for residents in Norway. The portal was introduced in 2011 [5]. In 2017 on average it had over 1.5 million visits per month [3]. Helsenorge is intended to be a single "point-of-entry" to electronic health services for residents. It consists of two parts, namely public and private. The public part is open for all and contains general information about diseases, treatments, patient rights etc. The private part of the portal requires authentication and contains individual's health related information. The total number of user sign-ins to the private part of Helsenorge doubled in 2017 compared to 2016 (from 3.2 million to about 7 million) [3]. Fig 1 shows a typical user interface of the system when a person is logged into Helsenorge [1].

An authenticated individual can see his or her health-related information such as the list of active and inactive medicines, history of hospital visits, verify a list of health-related payments, etc. He or she can also take an active part in one's health workflow via Helsenorge by performing various actions. For instance, an individual can change general practitioner; order, cancel or change doctor appointments; send request to renewal of medicine; send/receive messages to/from doctor or doctor offices; have a dialog with district health services; self-register own sickness; submit applications for reimbursement and so on. Although many of these electronic services are national, few of them are in pilot state, and therefore depend on the region where a person resides. Furthermore, a user has possibility to administer his or her own profile settings and

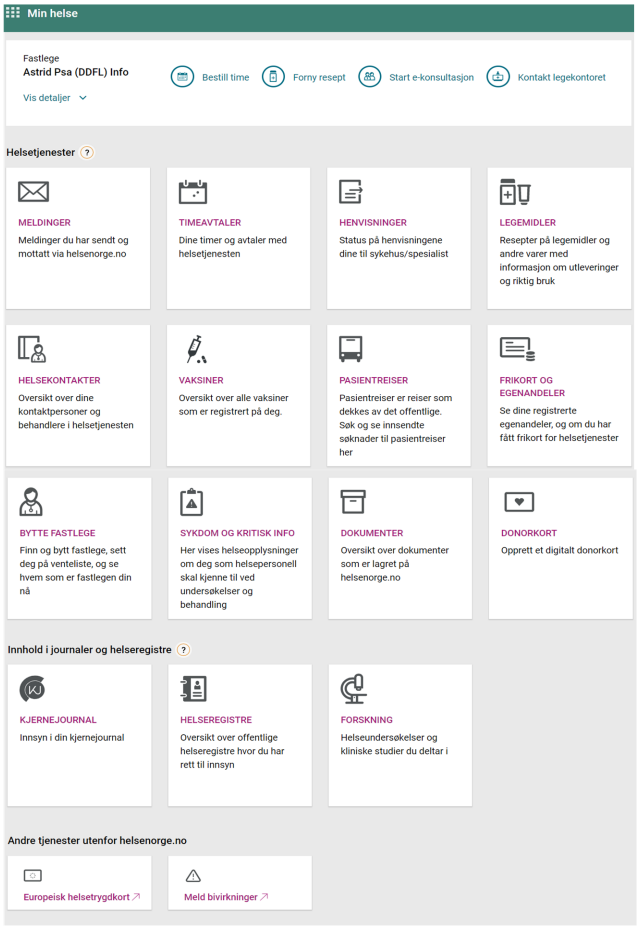

Figure 1: E-health services in Helsenorge.

access control levels. For instance, a user can agree or disagree for storing messages and documents in a personal health archive (PHA); restrict certain type of health personnel to access part of journal (Kjernejournal) information; decide not to use electronic services via Helsenorge anymore at any time (right to be forgotten), and so on. One important characteristic of the system is that users utilize e-health services not only on their own behalf but can also on behalf of their children, relatives or other person. This means one person can authorize access to another person to one or more predefined service areas in Helsenorge. For example, old parents can give access to their adult child to carry out a digital dialog with a health personnel on behalf of them via Helsenorge. Some services depend on the age and status of the users. Such access control flexibility comes at the cost of enhanced system complexity. In addition, medical and health information is regarded as a very sensitive and critical type of information. These all require more rigorous test and quality assurance activities of the system.

2.2 Helsenorge Release

Helsenorge has had four main releases per year excluding hotfix and other miscellaneous releases. However, from 2018 Helsenorge

[1]Language of the portal is Norwegian.

Table 1: Helsenorge's SCRUM teams

Team	Responsibility areas in Helsenorge
POT	Profile and access control
HOI	Health/medical information
TOD	Appointments and dialog
RØF	Reimbursements, economy
RÅI	Open part of Helsenorge

is expected to double number of main releases which consequently will increase test activity. Until introducing the test automation, significant amount of regression testbed on Helsenorge has been carried out manually. However, with increased number of software releases and testing iterations, achieving required level of test coverage with continuation of manual testing effort appeared to be unfeasible. We apply automation on Helsenorge's execution of regression test. Our test automation targets mainly private part of the portal Helsenorge.

2.3 Roles and Teams

We have defined two main roles within our automation concept, namely test automation engineer (TAE) who possesses programming skills and test analyst (TA) who does not need to have coding experience. Software development life-cycle model of Helsenorge follows the Agile methodology [2]. Currently, Helsenorge consists of five SCRUM teams which are listed in Table 1. Each team consists typically of a product owner, SCRUM Master, several programmers and one or two TAs (i.e. team testers). A TA is a full member of the development team with 100%-time dedication whereas TAE is not a full member of the team. He or she can be 50% in one team and 50% in another team, or dedicate 100% to one team temporarily. Currently at Helsenorge test group, the number of TAEs is less than the number of TAs. As of today there are 7 TA roles and 2.5 TAE roles.

Helsenorge development teams are self-organizing and cross-functional SCRUM teams. Self-organization means that the team itself decides how best to accomplish its work rather than being instructed by others outside the team [11]. Cross-functional team possesses all the skills that are necessary for the team to complete its tasks [11].

3 TEST AUTOMATION SOLUTION

3.1 Architecture

Figure 2 depicts generic test automation architecture (TAA) proposed by International Software Testing Qualification Board [1]. A drawback with this architecture is placing Test Data and Test Library components at the same level. We expect that managing Test Data component does not require programming skills if one has a domain knowledge, whereas for managing Test Script component, one must have programming knowledge. So, Test Data and Test Script components can be responsibility of TA and TAE, respectively.

We refine this generic TAA and propose new architecture for Helsenorge which is shown in Figure 3. As it can be seen from this

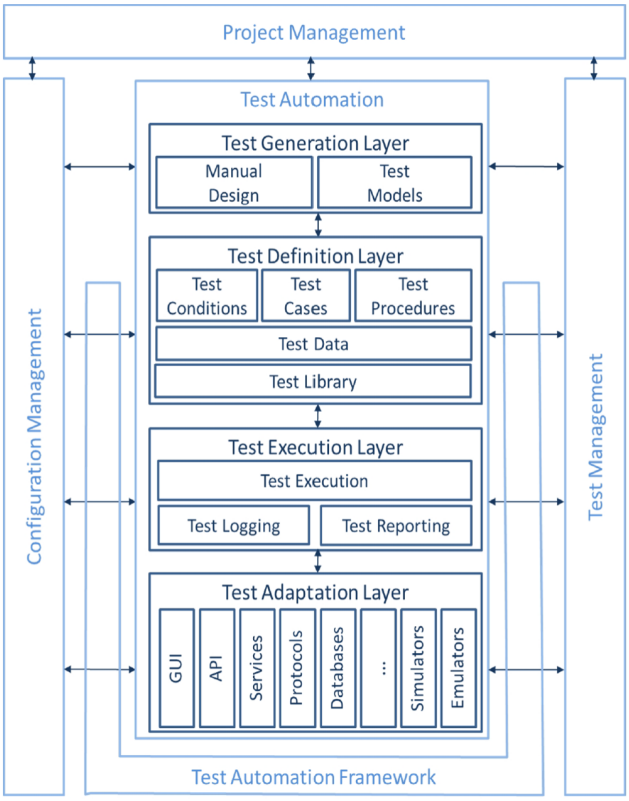

Figure 2: Generic test automation architecture by International Software Testing Qualification Board [1].

figure, in our architecture test generation layer is omitted since we manually carry out the test design and have not yet developed a model for Helsenorge. However, we introduce a new layer for test implementation to make easier separation of responsibilities between TAE and TA roles. In our architecture TA is responsible for test definition and test execution layers while TAE is responsible for test implementation and test adaptation layers. As it can be seen from the figures, unlike generic TAA in our architecture we moved Test Library from test definition layer to test implementation layer. This enables Test Data to be managed by TA whereas Test Library/Script to be maintained by TAE.

We use Microsoft's Team Foundation Server (TFS) as a configuration and test management tool. TFS is also used as a management tool for Helsenorge's requirements. This facilitates automatic traceability between a test case and a requirement. In other words, one can always determine which requirement is tested by which test case or vice-versa. More details of the architecture are described in subsequent subsections.

3.2 Test Implementation Layer

As mentioned earlier, TAE implements a test step to be executed automatically, and is responsible for maintenance of library of test scripts. An automated step is specified as a structured expression

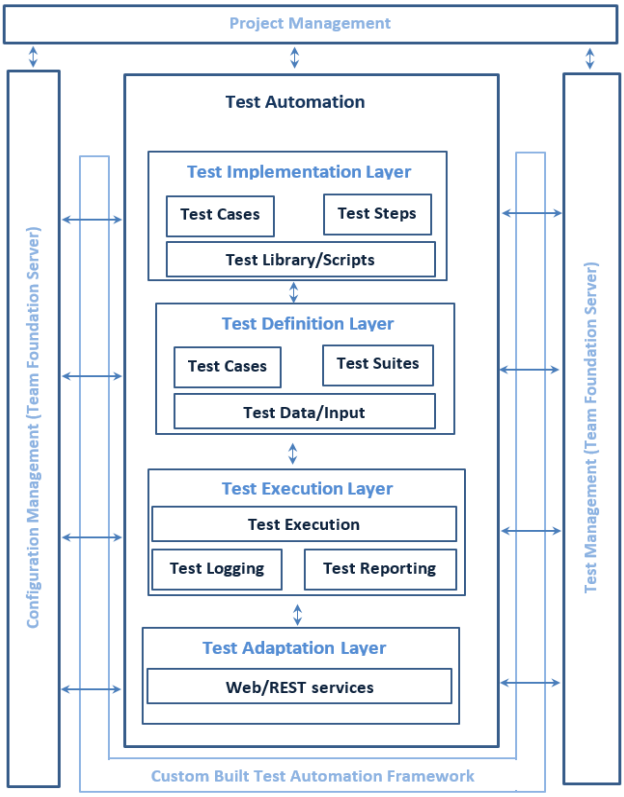

Figure 3: Helsenorge's Test Automation Architecture.

(non-free text) with or without parameters. The expression is formulated such that it is understandable by non-technical person with domain knowledge. Each automated test step is implemented as a C# class and bound via regular expression. Figure 4 shows an automated test step expression *Choose a representation "(.+)"* with its implementation code.

Storing standalone automated steps is not so meaningful therefore they are organized into ATC by TAE. We have defined two categories of ATC, namely template one and ordinary one. TAE creates and implements template ATCs that also serve as a repository of automated test steps. TA creates ordinary ATCs using automated test steps from the template ATCs.

3.3 Test Definition Layer

At test definition layer TA creates new ATC using automated test steps created in previous layer by TAE. Usually, conventional (manual) test cases specify expected outcome in each step. This enables TA to be aware what will happen after execution of each step and then compare actual and expected outcomes. However, in our ATC, we do not specify expected outcome in every step since we do not compare outcome in every step but rather have one explicit step for verifying expected outcome of the whole ATC. It is designed so because failure of comparison in a current test step ignores execution of the rest of the test steps in the ATC. In short, a single ATC verifies a single functionality. Furthermore, in our ATC, test

```
namespace Projects.Helsenorge.SharedSteps
{
    [BehavePatternStep("Choose a representation \"(.+)\"")]
    1 reference | Davrondzhon Gafurov, 66 days ago | 2 authors, 3 changes
    public class Representasjon : UserScenario
    {
        2 references | Bjørn-Vegard Thoresen, 214 days ago | 2 authors, 2 changes
        private string FulltNavn => GetTestDataOrValueToLower(1);
        0 references | Davrondzhon Gafurov, 291 days ago | 1 author, 1 change
        public Representasjon(IDependencyInjector injector) : base(injector)
        { }
        99+ references | Davrondzhon Gafurov, 66 days ago | 1 author, 2 changes
        public override async Task<TestStepResult> DoAsync()
        {
            await GetAndSaveResponseAsync(RestMethod.GetRepresentasjonsforhold);
            var res = await TestState.GetInstanceWithKeyAsync<HttpResponseMessage>(
                RestMethod.GetRepresentasjonsforhold.ToString());

            var hashed_pnr = await GetHashedPNR(res, FulltNavn);
            if (string.IsNullOrEmpty(hashed_pnr))
                return TestStepResult.Failed("Could not fint representation with a name '"
                    + FulltNavn + "'");

            var json = new JObject
            { ["Fnr"] = hashed_pnr };

            await PostAndSaveResponseAsync(RestMethod.SetRepresentasjonsforhold, json);
            await GetPageAsync(HelsenorgeArea.MinHelse);
            return TestStepResult.Successful();
        }
        1 reference | Davrondzhon Gafurov, 291 days ago | 1 author, 1 change
        private async Task<string> GetHashedPNR(HttpResponseMessage respone, string name)...
    }
}
```

Figure 4: Implementation of 2nd step in ATC shown in Figures 5 and 6.

input is an integral part of test case which helps TA to create more familiar (not very different from manual test cases), readable (in terms of business process) and self-descriptive automated test cases. In fact, the same test management tool (i.e. TFS) is used both for manual and automated test cases. Test inputs can be specified both as a value and as a variable. For example, in one ATC a test step can be inputted with value while in another ATC the same test step gets input from variable. Figures 5 and 6 show two ATCs [2] where the test step *Given that I logged in as …* has input as a variable *@pnr* and as a value *31026600000*. TA themselves can choose which way they want to specify input, e.g. depending on repeatability of the input within a given ATC. ATC in Figure 5 iterates twice with two sets of test input, while ATC in Figure 6 iterates (runs) once. In general, an ATC can be iterated with any number of test inputs.

3.4 Test Execution Layer

TA is responsible for arranging set of ATCs into test suites, executing them and observing test execution report. Test suites are executed daily at specified time or can be triggered on demand by TA. However, the task of failure analysis is a shared responsibility between TA and TAE. Source of failure can be different: defect in SUT (primary we are interested in), error in test environment (e.g. a service is down), outdated test script or test data and so on. If execution of ATC fails then at first TA attempts to analyze and investigate failure. If TA cannot determine the failure reason, only then TAE starts failure analysis and investigation, see Figure 7 for this workflow. By letting first TA to analyze and possibly partially or fully resolve failures, we avoid TAE to be a "bottle-neck" in the

[2] Test cases are specified in Norwegian language. Here they are translated into English for understandability purpose.

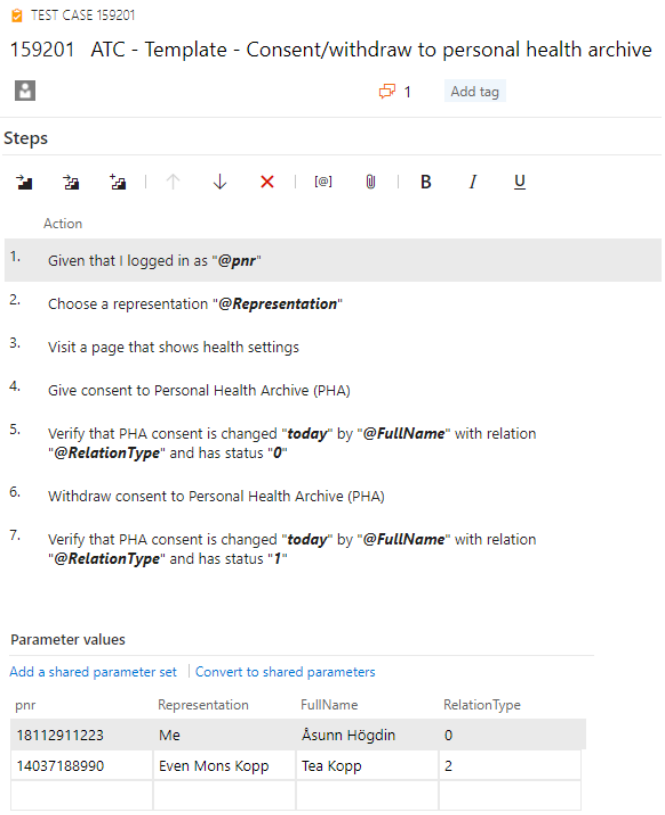

Figure 5: Example of the automated test case.

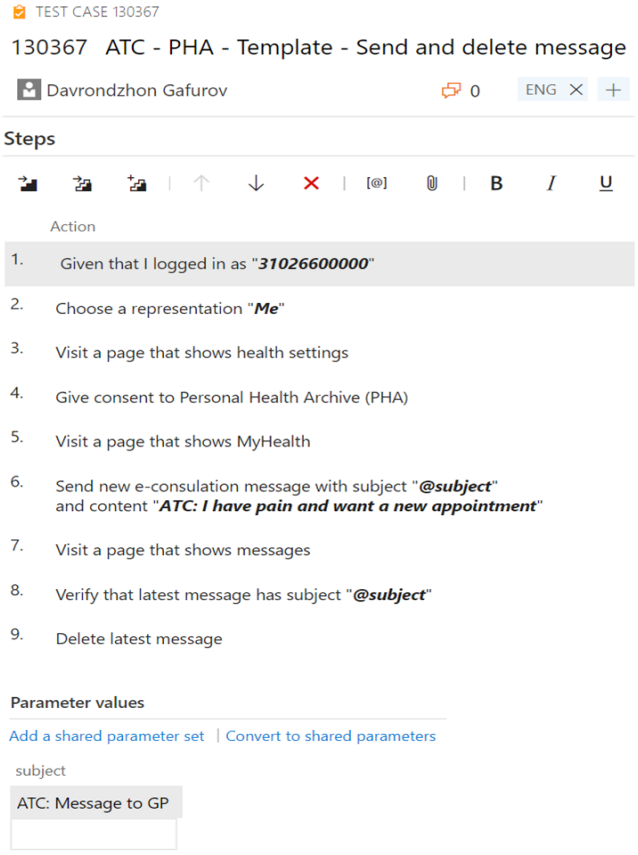

Figure 6: Example of another automated test case.

process. For TA to analyze independently it is important to report meaningful error messages when a failure occurs.

3.5 Test Adaptation Layer

At this layer TAE implements test interface at which ATC shall interact with Helsenorge. Once this layer is completed, usually it does not require further modifications unless new interaction level is necessary to implement. Most of the manual test cases that TAs have created run at Graphical User Interface (GUI) level. However, we have chosen to apply test automation at the REST (REpresentational State Transfer) service [7] level, since elements on GUI tends to be changed frequently. Although our approach may not detect GUI specific bugs, it provides opportunity for TA to create ATC that can discover back-end defects by bypassing front-end validation. TA can create the so called negative ATC which tests SUT with unexpected (illegal or incorrect) inputs and verifies relevant error messages are reported. Such back-end defects are difficult or impossible to discover by manual GUI tests. For instance, in Helsenorge portal the valid health region codes are 1 and 2. Figure 8 shows an example of negative ATC which attempts to set the region code to some invalid values (i.e. 11, -2 and 0). The last step in this ATC verifies that the correct error message is reported. This test is not possible to run manually because GUI does not allow the user to select the values other than 1 or 2 for health region.

4 TEST CREATION EFFICIENCY

4.1 Automation Phases with and without Test Artifacts

In terms of measurable test artifacts, work on the test automation can be divided into two phases, the ones that generate (explicit):

(1) Measurable test artifacts
(2) Non-measurable test artifacts.

In our architecture from Figure 3 an automation work on the test implementation, test definition and test execution layers produce measurable test artifacts. It means at these layers one can ask questions such as, how many automated test steps, automated test cases or test execution reports are implemented, created, or produced, respectively. However, the automation work on test adaptation layer does not produce any direct test artifact. A test adaptation layer produces a test code that enables interaction with SUT but no explicit test artifacts that TA can reuse or work with.

4.2 Test Creation Efficiency

As noted previously, to create a new ATC the TA uses automated test steps from the template ATCs. Although knowing how many ATCs have been created by TAE or TA is important for monitoring work progress, it would also be useful to know the impact of TAE's

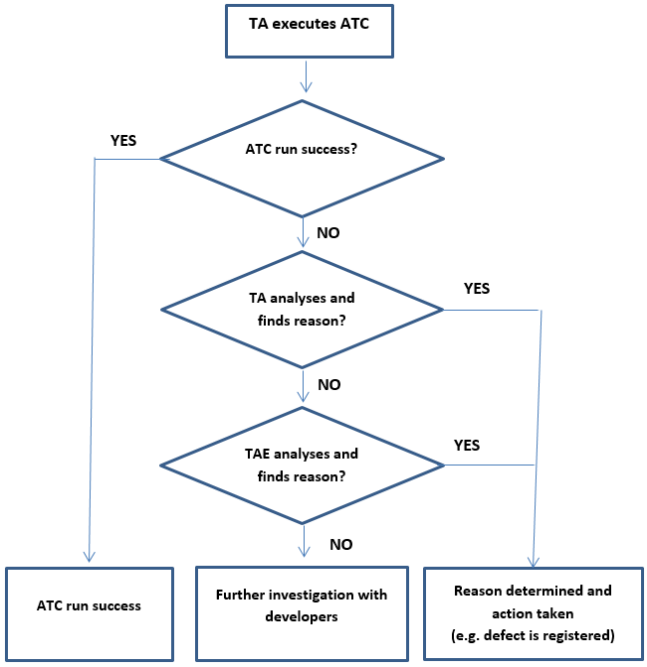

Figure 7: Helsenorge's defect investigation process.

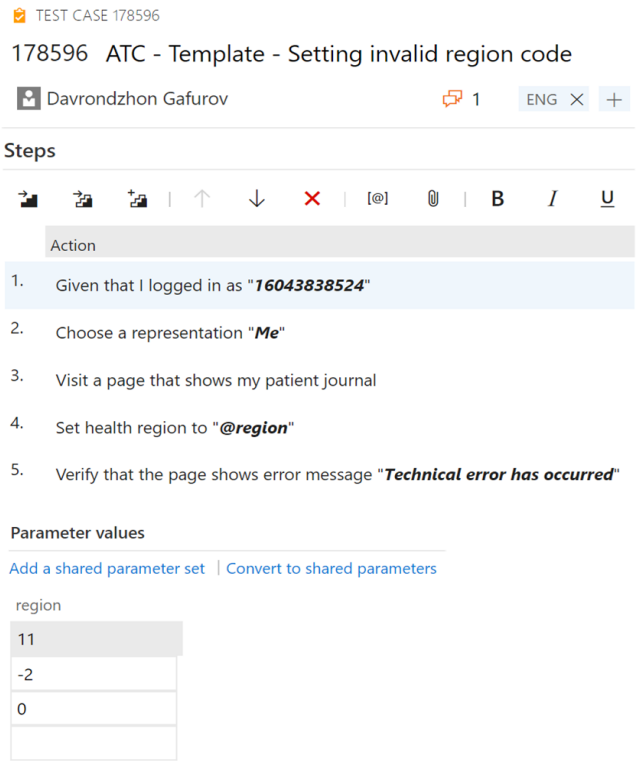

Figure 8: Example of a negative automated test case.

Table 2: Test creation efficiency for Helsenorge teams

Team	ATC by TAE	ATC by TA	Total	R
POT	33	41	74	1.24
HOI	46	2	48	0.04
TOD	12	2	14	0.17
RØF	3	0	0	0
RÅI	0	0	0	NA
Total	94	45	139	0.48

work on TA's work. For instance, how many new ATCs the TA can create based on the existing template ATCs created by TAE. In other words, checking whether it is possible to predict TA's work based on TAE's work. For answering this, we define the *test case creation efficiency ratio* as follows

$$R = \frac{Total\ \ ATC\ \ created\ \ by\ \ TA}{Total\ \ ATC\ \ created\ \ by\ \ TAE} \quad (1)$$

Table 2 shows Helsenorge's test case creation efficiency ratio per team. The high number for POT team is due to the fact that the core functionality is developed by this team, and the automation project started with this team. In fact, teams in Table 2 are ordered in chronological order of using test automation i.e. we applied our solution first on POT and last on RØF. For testing activity in RÅI team (open part of the Helsenorge) our automation solution is not applied yet. As of today, the total test creation efficiency ratio for Helsenorge is 0.48. It means if TAE created 10 ATCs then for TA it was possible to increase the coverage by creating and adding 4 or 5 new ATCs based on this 10. However, it should be noted that this number reflects only creation (and deletion) of new test cases but does not considers modification or update of existing test cases. For example, if TA adds new test input to already existing ATC, then it is not reflected in this number.

5 BENEFITS, LESSONS LEARNED AND LIMITATIONS

The work on designing, architecture, implementation and deployment of test automation solution for Helsenorge started early in 2017. Consequently, benefits, limitations, lessons and findings presented in this paper are those we have experienced so far. However, we plan to observe and collect further experiences and findings which we will report in future publications.

5.1 Benefits

The followings are the main benefits we achieved using our approach:

- Saved time and improved coverage. By applying this automation, the greatest benefits that we achieved were, not surprisingly, saved manual regression testing effort and improved test coverage. The automation allowed us to carry out not only breadth but also depth testing. We refer to the breadth testing when we have several ATCs that verify separate features of the SUT. By depth testing we mean a single ATC is iterated with several test data. For example, there are five

Helsenorge's service areas which are available via authorization (to give access to another person). Test scenarios of all possible combinations of this feature are 32. A single ATC is created to iterate over all these 32 combinations. According to the TA in POT team, it takes about 5 minutes for her to manually test one iteration of this feature. However, automatic execution of one iteration takes about 14-15 seconds.

- Better cross-functional teams. Another important benefit of our approach was its contribution towards better cross-functional development teams. Cross-functional team has all necessary skills to accomplish its task without depending on others not part of the team [11]. In case of testing tasks, TA (who is a full member of the development team) can be also in charge of test automation because independently he or she can
  – create a new ATC,
  – update existing ATCs,
  – organize ATC into test suites,
  – manage execution of test suites and analyze test report.
  Thus, TA's automation work does not depend on TAE's work as long as required test steps are automated (i.e. implemented by TAE). In addition, being able to independently carry out a test automation may help TA in better estimating testing tasks during work (sprint) planning.
- Negative tests. As it was pointed out, our approach enables creating negative ATC with illegal or incorrect test inputs that verify fail-safe scenario of the system (see for example Figure 8). In fact, we plan to increase the number of negative ATC as we believe many back-end defects can be detected by bypassing front-end validations which are not feasible via manual GUI tests.

5.2 Lessons Learned

The following is a short summary of the main lessons learned from our experience:

- Understand before code. TAE should have a holistic perspective to design and implement efficient (reusable) automated test steps. He or she needs to have a good understanding of functionality and interdependencies among various parts of the SUT before commencing to write a test code. It is also important a choice of language formulation for specification of a test step that adequately reflects SUT functionality. This is especially true for frequently used test steps which are shared and used by several teams. Otherwise, it may result in several implementation iterations or code duplication for the same or similar functionality. Therefore, good collaboration between TAE and development teams, especially TAs, is essential.
- Always have maintenance perspective. Almost every test task can be automated and test managers may have high expectations with respect to automation. However, test automation is not a one-time job, it is a continuous process and therefore maintainability is essential. In fact, maintenance of ATC can be challenging and failure source of many test automation projects [6, 10]. Consequently, TAE shall evaluate every test automation task not only from the implementation perspective (short-term goal) but also in terms of maintainability (long-term goal). If maintenance effort of the task appears to outperform its automation benefits then it shall not be automated. Our approach enabled delegating part of maintenance effort to TA (e.g. updating ATC, managing test data). This is important because of minimizing risk of "bottle-necks" in the process due to limited number of TAE resources in the organization. In addition, to have effective maintenance one must create guidelines both for TAE and TA, for example on how ATCs will be implemented, created, updated and eventually removed; and criteria for selecting test tasks for automation.
- Inform managers about non-measurable phases. Project and test managers shall be informed and aware about automation phases that do not produce explicit measurable test artifacts. Otherwise, they may get a wrong impression that the test automation work is not progressing as expected.
- Tool unification. It is desirable to have the same management tool for both manual and automated test cases as well as for requirements. This simplifies automatic generation of various test execution and test coverage reports. Otherwise, one needs to estimate additional work for TA to (manually) synchronize/combine test reports for manual and automated test execution from two separate sources and maintain traceability between test cases and requirements (possibly) manually.

5.3 Limitation and Opportunities for Future Work

Below is the list of some main limitation and opportunities for future work:

- Tool limitation. Several limitations that we came across during implementation were intrinsic to the test management tool i.e. TFS. One of them was not being able to control execution order of the ATCs within a test suite otherwise one could create more compact test suites to achieve the same level of test coverage. Another functionality that the tool could provide is an automatic suggestion of test steps while TA typing the first letters or words of the test case. This could help to avoid false negatives due to spelling errors in the test step language.
- Automated test data generation. In our approach, TA manually updates ATC with test data. Although manually updating or editing test input in an ATC is easy and straightforward, it can be tedious task when the number of ATCs to update is large. In addition, it may take time for TA to find test data with the required characteristics (e.g. finding a test person which has a child with digital GP or electronic prescription). We investigate possibility to automate updating test data in ATC i.e. automating test data generation.
- Automated test case generation. In our approach, creation of automated test cases is performed manually. If SUT is modified in a way that requires a modification of existing ATCs then it creates extra work for TA to edit or even recreate them manually. However, availability of SUT model

from where ATCs can be generated automatically will reduce this work. For instance, after SUT changes, only its model is updated and then test cases are generated automatically from the model. We plan to investigate possibility to develop a model of Helsenorge for automated generation of test cases.

6 CONCLUSION

We proposed a test automation solution which was implemented by using process driven test automation. It has been applied and still in operation on a large web application of electronic health services in Norway (Helsenorge.no) which has over one million visits per month. Our solution enabled moving (part of) test automation tasks from test automation engineer with coding skills (expensive resource) to test analyst with domain knowledge (less expensive knowledge). Besides expected benefits of saved manual testing effort and improved test coverage, another important advantage of our approach is that it contributed towards better cross-functional development teams. This enables a test analyst to independently carry out necessary test automation activities. Such activities include creating new or updating existing automated test cases, executing them, and analyzing test report. An important lesson we have learned is that a test automation engineer needs to have a good understanding of functionality and interdependence of the system to implement efficient automated test steps (i.e. understanding before coding). In addition, the test automation engineer should always remember maintainability aspect while considering automation of the test task.

As of today, Helsenorge has 139 automated test cases, and test creation efficiency ratio is 0.48 which means almost 1/3 of 139 test cases are created by test analysts while the rest is created by the test automation engineer. Our experience indicates promising results, and we plan to apply the approach on other products and development teams within our organization.

REFERENCES

[1] International Software Testing Qualifications Board. [n. d.]. Test Automation Engineer – Advanced Level Syllabus, version 2016. Retrieved February 3, 2018 from www.istqb.org

[2] Torgeir Dingsøyr, Sridhar Nerur, Venu Gopal Balijepally, and Nils Brede Moe. 2012. A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and Software* (2012).

[3] Direktoratet for e helse. 2018. Utviklingstrekk 2018 - Beskrivelser av drivere og trender relevant for e-helse. Report on trend relevant for e-health. Report is in Norwegian.

[4] Vahid Garousi and Frank Elberzhager. 2017. Test Automation: Not Just for Test Execution. *IEEE Software* (2017).

[5] Helsenorge.no. [n. d.]. Helsenorge.no

[6] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. 2010. Software Test Automation in Practice : Empirical Observations. *Advances in Software Engineering - Special issue on software test automation* (2010).

[7] Li Li and Wu Chou. 2011. Design and Describe REST API without Violating REST: A Petri Net Based Approach. In *IEEE International Conference on Web Services*.

[8] Mark Micallef and Christian Colombo. 2015. Lessons learnt from using DSLs for automated software testing. In *IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.

[9] Macario Polo, Pedro Reales Mateo, Mario Piattini, and Christof Ebert. 2013. Test Automation. *IEEE Software* (2013).

[10] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *International Workshop on Automation of Software Test (AST)*.

[11] Ken Schwaber and Jeff Sutherland. 2017. The Scrum Guide.

[12] ISO/IEC/IEEE 29119 Software Testing. 2016. ISO/IEC/IEEE 29119-5: Keyword-Driven Testing.