Efficiency Metrics and Test Case Design for Test Automation

Davrondzhon Gafurov and Arne Erik Hurum
Norsk Helsenett SF
Oslo, Norway
davrondzhon.gafurov@nhn.no

***Abstract*—In this paper, we present our test automation work applied on national e-health portal for residents in Norway which has over million monthly visits. The focus of the work is three-fold: delegating automation tasks and increasing reusability of test artifacts; metrics for estimating efficiency when creating test artifacts and designing robust automated test cases. Delegating (part of) test automation tasks from technical specialist (e.g. programmer - expensive resource) to non-technical specialist (e.g. domain expert, functional tester) is carried out by transforming low level test artifacts into high level test artifacts. Such transformations not only reduce dependency on specialists with coding skills but also enables involving more stakeholders with domain knowledge into test automation. Furthermore, we propose simple metrics which are useful for estimating efficiency during such transformations. Examples of the new metrics are implementation creation efficiency and test creation efficiency. We describe how we design automated test cases in order to reduce the number of false positives and minimize code duplication in the presence of test data challenge (i.e. using same test data both for manual and automated testing). We have been using our test automation solution for over three years. We successfully applied test automation on 2 out of 6 Scrum teams in Helsenorge. In total there are over 120 automated test cases with over 600 iterations (as of today).**

***Index Terms*—Test automation, automation measurements and metrics, Helsenorge**

I. Introduction

The increasing complexity, pervasiveness and interconnection of software systems on the one hand, and the ever-shrinking development cycles and time-to-market on the other, make the automation of software test an urgent requirement today more than ever. A primary aim of test automation is minimizing the manual testing effort and reducing overall product development cost. However, despite significant achievements both in theory and practice, test automation remains a challenging task [11], [14]. If not implemented properly, it can be expensive and even contribute to the increase of the cost.

This paper presents our experiences of using test automation for over three years. One of the challenges (which we have also faced) in test automation was finding talents with the right technical skills and testing mindset. To address this challenge, we developed a solution that enables delegating part of automation tasks to the tester without coding experience. Delegating (part of) test automation tasks from technical specialist (e.g. programmer - expensive resource) to non-technical specialist (e.g. domain expert, functional tester) is carried out by transforming low level test artifacts into high level test artifacts. Such transformations not only reduce dependency on specialist with coding skills but also enable involving more stakeholders with domain knowledge into test automation. Furthermore, we propose simple metrics which are useful for estimating efficiency during such transformations. Examples of the new metrics are implementation creation efficiency and test creation efficiency. We describe how we design automated test cases in order to reduce the number of false positives and minimize code duplication in the presence of test data challenge (i.e. using same test data both for manual and automated testing).

The rest of the paper is structured as follow. Section II gives a brief overview of Helsenorge which is our system under test (SUT). Section III describes test automation artifacts, roles and our test automation architecture. Section IV presents our proposed empirical metrics. Section V discusses designing robust automated test cases. Section VI summarizes main benefits, practical recommendation and limitations and opportunities for future work. Section VII concludes the paper.

II. Helsenorge - System Under Test

Helsenorge is a national portal of e-health services for residents in Norway which was introduced in 2011 [10]. In 2018 Helsenorge had 25.6 million visits while in 2017 it had 18.7 million visits [4]. 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 to everyone 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. In 2018 the total number of user sign-ins to the private part of Helsenorge was 12 million while in 2017 it was about 7 million (and 3.2 million in 2016) [4].

An authenticated individual can see and verify his or her health-related information such as the list of medicines, vaccination history, hospital visits, 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, with few clicks an individual can become an organ donor; order, cancel or change doctor appointments; send request to renewal of medicine; have a dialog with district health services; self-

978-1-7281-8915-4/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS-C51114.2020.00015

register own sickness; submit health related travel expenses for reimbursement and so on.

III. ARTIFACTS, ROLES AND ARCHITECTURE

A. Test artifacts

Test suites, test cases, test steps, test implementation code and test data are all examples of test artifacts. A *test suite* is a collection of test cases organized based on some characteristics, e.g. running environment, test type, target area of SUT etc. A *test case* is an ordered sequence of test steps that verifies a (single) functionality of SUT. A *test step* is an instruction that is executed to affect (write action) or to verify (read action) state of SUT. An automated test step is the one that is executed automatically. In order to make a manual test step to run automatically an *implementation code* has to be written. An automated test case is the one where all of its test steps are automated. We use the terms ”test case” and ”automated test case” interchangeably, both referring to automated test case. Similarly, we use terms ”test step” and ”automated test step” interchangeably, both referring to automated test step. When referring to the manual test step or manual test case, we state it explicitly. A test step can be executed with one or more *test data* (test inputs). When a test case is executed with n different test data then we say that the number of iterations for the test case is n. A test case has at least one iteration.

B. Test roles and delegation

The tasks related to test automation, we divide into two groups the one that requires programming skills and the other one where domain knowledge is sufficient (and coding skills is not mandatory). For example, to create an implementation code that makes a manual test step to execute automatically requires programming skills. However, composing a test case using existing test steps or updating the test case with a new test data does not require coding expertise as long as one is familiar with SUT functionality. Within our test automation solution, we have two main roles *Test Developer (TD)* and *Functional Tester (FT)*. TD’s main responsibility is to convert manual test steps (test cases) into automated counterparts by writing implementation code. FT’s main responsibility is to create test cases using automated test steps, update them (when necessary) and run test cases. The role of TD requires programming expertise while the role of FT does not require technical experience. In short, TD is responsible for all automation tasks that require programming expertise while FT for the tasks that do not require coding skills. As of today Helsenorge consists of six Scrum teams. Each team has ScrumMaster, Product Owner, several developers and at least one dedicated FT. However, Scrum teams do not have 100% dedicated TD role.

By delegating part of automation tasks to FT we aim to reduce dependency on TD and avoiding bottleneck situations since the role TD is more expensive and sparser compared to the role of FT (based on our own experience too). In general, delegation allows not only FT but also other team members (e.g. Test Manager, Product Owner) who has domain knowledge but not necessarily programming skills to contribute in to automated testing.

C. Test architecture

Delegation requires appropriate test architecture and techniques. Figure 1a depicts a generic Helsenorge test automation architecture which is based on the architecture proposed by International Software Testing Qualification Board (ISTQB) [3]. The key difference is that we move Test Library component from Test Definition layer to new Test Implementation layer [9]. We believe that managing Test Data component does not require programming skills if one has a domain knowledge, whereas for managing Test Library component one must have programming knowledge. Having Test Data and Test Library on their own layers facilitates separation of duties between TD and FT roles.

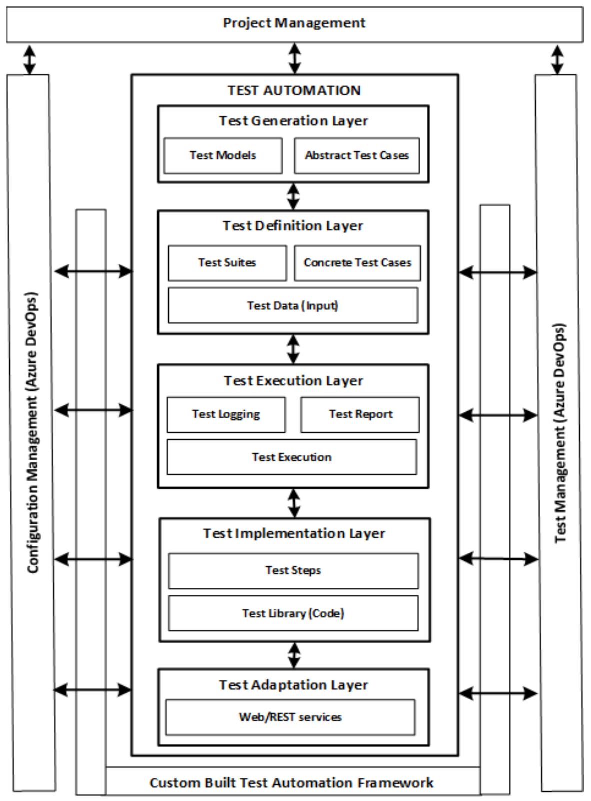
We use Microsoft’s Azure DevOps (ADO) both as the test and requirement management tool for Helsenorge. Both manual and automated test cases are created, updated and stored in ADO. We also developed a custom automation framework using Microsoft’s Visual Studio (VS) .NET C#. The framework serves as a bridge between Test Library component and test management system. This provide us opportunity to fully utilize test reporting features of ADO, for example not only showing current (or last) execution but also historical results. Tool unification enables automatic traceability between the test case and its requirement which is important criteria. In other words, one can always determine which requirement is tested by which test case and vice-versa.

D. Test adaptation and implementation

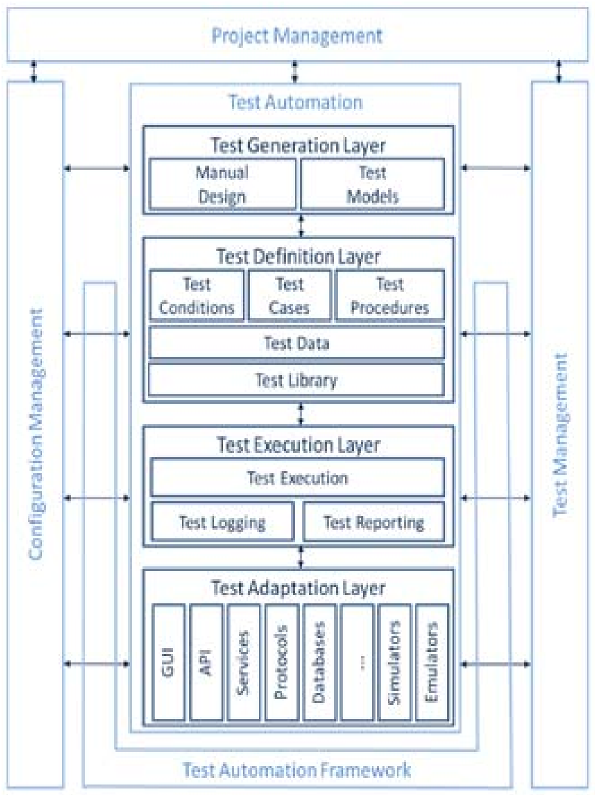
Test automation is applied at the REST (REpresentational State Transfer) API level [12], since elements on GUI tends to be modified frequently. Test implementation code is written using C# programming language in Visual Studio (VS). Our test automation belongs to keyword- or process- driven techniques [3], [13]. Each test step serves as a keyword and executed automatically. Test step’s definition text in ADO and its implementation in VS is bounded via regular expression. Figure 2 shows part of the implementation code for the test step 2 in Test case 1. An automated step is specified as a structured expression which is plain text and formulated such that it is understandable by non-technical person with domain knowledge. A tester with domain knowledge can compose a test case by using existing test steps.

E. Test execution

FT is responsible for arranging test cases into test suites, executing them and observing generated test report. Test suites are executed periodically at specified interval or can be triggered on demand by FT. When test case execution fails, 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. The task of

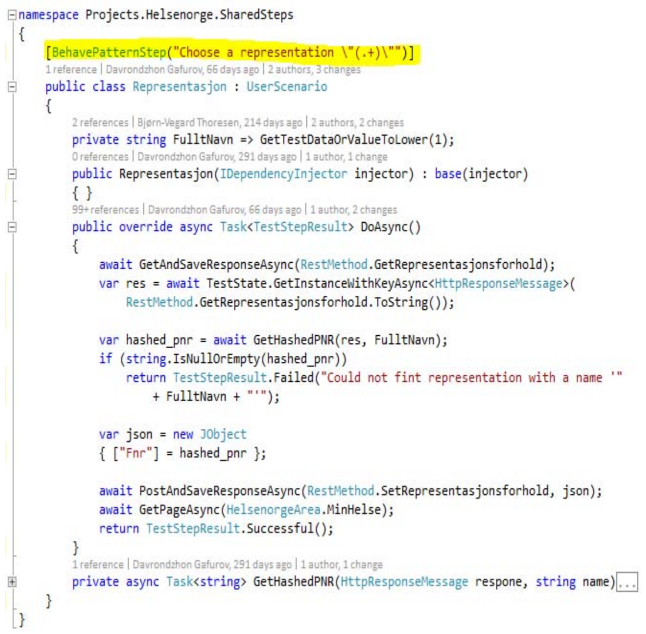


(a) Helsenorge's Test Automation Architecture.



(b) Generic test automation architecture by ISTQB [3].

Fig. 1: Helsenorge and ISTQB Test Automation Architecture.



```
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)...
    }
}
```

Fig. 2: Implementation code snippet.

failure analysis is a shared responsibility between FT and TD. At first FT attempts to analyze and investigate failure based on generated test report and provided error messages. If FT cannot determine the cause of the failure, only then TD starts failure analysis and investigation. By letting first FT to analyze and possibly partially or fully resolve failures, we avoid TD to be a "bottle-neck" in the process. For FT to be able to analyze failure independently it is very important to report meaningful error messages.

Test generation layer in terms of model-based testing of our work can be found in [7], [8]. The test definition layer is described in Section V.

IV. Measurements and metrics

A. Abstraction levels

Figure 3 presents layered view of our test automation where test artifacts are transformed from low level abstraction to high level abstraction. Abstraction level of test artifacts determine the required level of technical competence. Artifacts at the low level abstraction require coding expertise, while for artifacts at the high level abstraction domain knowledge is sufficient. Test artifacts at the levels 1, 2, 3 and 4 are C# classes (implementation units), test steps, test cases and test suite(s), respectively. Test data also belongs to the level 3. As can

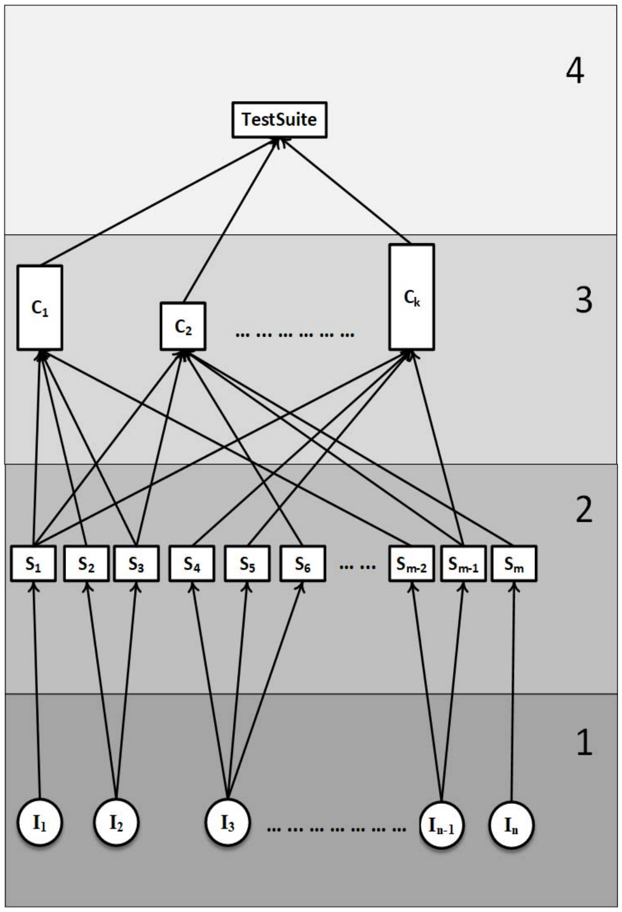


Fig. 3: Layered view on test artifacts. Varying heights of test cases illustrate different number of iterations.

be seen from Figure 3, each test step has a corresponding implementation unit which makes it run automatically. Several test steps can share the same implementation unit (one to many mapping). A test step can be re-used in many test cases or even multiple times in the same test case (one to many mapping). Example of test case.

Transitions of artifacts from level 1 to 2 is responsibility of TD because creating implementation units requires programming skills. However, TD should consult with FT in order to define appropriate text for the test step. It is very important that test steps are understandable by FT, since they are the primary end-users of the test steps. Transitions of test artifacts from level 2 to 3 and from 3 to 4 are responsibility of FT and do not require coding skills. The only prerequisite for these transitions is the knowledge of SUT and availability of test data. Consequently, transitions from 1 to 2 is more expensive compared to transitions 2 to 3 or from 3 to 4. Table I presents abstraction level, test artifacts, who is main responsible and the tool used to work with artifacts.

TABLE I: Test artifacts at abstraction levels

Level	What	Who	Tool	Level
Level 4	Test suites	FT	ADO	Product
Level 3	Test data	FT	ADO	System
Level 3	Test cases	FT	ADO	System
Level 2	Test steps	TD/FT	ADO	Domain
Level 1	C# classes	TD	VS	Implementation

TABLE II: Test code statistics

Parameter	Value
Lines of code	25400
Maintainability Index	78
Cyclomatic complexity	4092
Depth of Inheritance	6
Class coupling	726

B. *Efficiency metrics*

Although measurement like the number of test cases, test steps, implementation units are necessary to show progress and coverage, the metrics showing their relations can give more insights. We define *Implementation Creation Efficiency (ICE)* metric as follow:

$$ICE = \frac{m}{n} \tag{1}$$

where n is a total number of C# classes (i.e. implementation units), and m is a total number of test steps (i.e. implementation representations). The ICE is estimated when test artifacts are transformed from the abstraction level 1 to the level 2. The ICE indicates how ”good” high level artifacts (i.e. test steps in this case) can be represented by low level artifacts (i.e. implementation units - C# classes). In general, the higher the ICE the better it is i.e. ”many” test steps (which is used by FT) are generated from ”fewer” C# classes (which are produced by TD). However, too large ICE can also indicate problems. In theory, one can have one or few large implementation units for all representations. This means complex code base which is difficult to maintain. Therefore, one has to set a certain criteria for individual implementation units in order to limit code complexity. Examples of such criteria can be lines of code (LOC), cyclomatic complexity, etc. The total LOC of automation solution for Helsenorge is 25.400 excluding the packages which are responsible for interaction with test management system, see Table II.

Our C# classes have requirement as maximum cyclomatic complexity of 5. The lowest number for ICE is 1 which essentially means one representation for one implementation. The optimal value for ICE varies depending on SUT but it is certainly neither too large nor 1. The ICE can also indicate the level of understanding of SUT by TD. The good number of ICE is the indication of that TD has holistic perspective of SUT and not merely focuses on writing code. To achieve good number on ICE we recommend that TD (especially junior TD) should have FT role for some period when he or she joins the team before starting to write the test code. In this way TD will get better understanding of SUT which is a good starting point to write efficient code (and hopefully optimal ICE).

We also define *Test Creation Efficiency (TCE)* metric as follow:

$$TCE_1 = \frac{k}{m} \qquad TCE_2 = \frac{k}{n} \tag{2}$$

where k is a total number of test cases and n and m as in formula 1. TCE_1 is based on number of test steps while TCE_2 is based on number of implementation units. When referring to both TCE_1 and TCE_2, we will use TCE. The TCE is estimated when test artifacts are transformed from the abstraction level 2 to the level 3. The TCE indicates possibility of creating test cases based on the given number of lower level test artifacts. An interesting point about TCE is that it can be used to estimate or predict (or at least give an indication of) the workload of FT based on the work of TD. For example, how many implementation units is necessary to create one test case for the given SUT. The larger TCE the better it is.

At the level 4 (in Figure 3) total number of test cases in a test suite and ratio of automated test cases to the manual ones are often used as a measurement. However, sometimes these measurements are not enough to provide full picture of coverage. For instance, when a different part of SUT has varying degree of risks and require unequal degree of test coverage. The total number of iteration is more complete measure of coverage criteria. In this respect, we define *Average Depth Coverage (ADC)* metric as follow:

$$ADC = \frac{d}{k} \tag{3}$$

where d is a total number of iterations and k as in formula 2. ADC indicates the average number of iterations a test case is executed or how good is the test coverage in depth. In addition to ADC, one may also estimate *Variance of Depth Coverage* (VDC). The large variance may indicate the variable degree of criticality of SUT because that some part of SUT is required to be tested more deeper than other parts. The minimum number for ADC is 1 which means every test case is run once (one iteration). Although ADC can also be estimated for the manual testing, usually manual test cases are restricted to limited number of iteration due to being time-consuming and requiring tedious manual effort. However, once a test case is automated it relatively cheaper to execute it with more test data (input) to increase depth coverage and obtain better quality assurance. Therefore, it is more cheaper to increase ADC with automated testing compared to with manual testing. Consequently, ADC can also indicate how "good" test automation is applied. ADC shall be used as a complimentary metric to total number of test cases.

C. Automation measurements and metrics

Helsenorge consists of six SCRUM teams and we have applied our automation solution on two of them. Those two Scrum teams are so-called POT and HOI. The team POT is responsible for privacy and access control while the team HOI for information and services related to the individual's health. Table III presents ICE, TCE and ADC metrics and other automation measurements for Helsenorge [1]. Since implementation units and unique test steps are re-usable across teams (test suites), we do not assign them to the specific teams (i.e. first two rows in Table III for POT and HOI are empty). There are various factors which influence these numbers such as SUT complexity, integration, skills of TD and FT, etc. It should be also noted that there are several helper C# classes which are not taken into account while calculating ICE and TCE in formulas 1 and 2 because they relate more to the general framework rather than individual artifacts. As can be seen from the numbers in Table III, POT team has better coverage compared to the team HOI. This is due to the fact that we began test automation with POT team, and it is the team which provides services to all other teams in Helsenorge.

TABLE III: Automation measurements and metrics

Measurement	*POT*	*HOI*	*Total*
Number of C# classes, n	-	-	133
Number of unique test steps, m	-	-	215
Number of test steps in test suite, r	4284	433	-
Number of test cases in test suite, k	80	46	126
Number of test iterations in test suite, d	568	67	635
Execution time, approx. minutes	51	8	59
Metric	*POT*	*HOI*	*Total*
ICE, m/n	-	-	1.62
TCE, k/m	0.37	0.21	0.59
ADC, d/k	7.10	1.46	5.04
Test step reusability, r/m	19.9	2.0	21.9
Average test steps per iteration, r/d	7.5	6.5	7.4

V. Designing automated test cases

A typical structure of the manual test case is that every (manual) test step has corresponding expected result. While running the test case manually, FT manually (or visually) verifies that the expected outcome matches the actual outcome after each test step execution. Sometimes during execution FT may skip running some steps due to the fact that the required state of SUT has already been made by other tester(s). In other words, FT can fix the precondition for the test case "on fly" and re-run test steps in the manual test case.

Traditionally the expected result of the manual test step is converted to assert in automated version within implementation unit of the corresponding test step. That is fine as long as test data for automated and manual testing are separate. And for optimal results in automation, test data for manual and automated testing shall be separate. However, this is not the case in our SUT.

One of our main challenges is using the same test data for manual and automated testing. This is due to the fact that our test data have many attributes which come from various external systems. Helsenorge (our SUT) obtains health related data from many external sources. Therefore, it is costly and difficult (if not impossible) to have separate sets of test data for manual and automated testing. When a test case is executed

[1]The numbers are "as of today".

with a test data, we don't know its state in advance, and asserting in such situations often fails. For example, all our test cases require that a user has a certain consent level, and when a test case is executed the consent level of the user is unknown beforehand. Therefore, automated test cases are run with zero or minimal assumption about the state of test data.

TEST CASE 1: Creating authorization - all possible combinations of authorization areas

1 Given that I logged in as ***"30029011111"***
2 Choose a representation ***"Myself"***
3 Visit a page that shows authorizations
4 Delete all given authorizations
5 Visit a page that shows settings
6 Set consent level to ***"3"***
7 Visit a page that shows authorizations
8 Create authorization with these parameters: ***"91028822222"***, ***"Lee"***, ***"@Area"***,***"2030-11-11"***
9 Verify that I gave authorization with these parameters: ***"91028822222"***, ***"Lee"***, ***"@Area"***,***"2030-11-11"***
10 Given that I logged in as ***"91028822222"***
11 Choose a representation ***"Myself"***
12 Visit a page that shows authorizations
13 Verify that I received authorization from ***"30029011111"*** to ***"@Area"*** which is valid until ***"2030-11-11"***
14 Given that I logged in as ***"30029011111"***
15 Choose a representation ***"Myself"***
16 Visit a page that shows authorizations
17 Delete authorization given to ***"91028822222"***

```
// Test parameters
```

Area
8
9
10
8,9
8,10
9,10
8,9,10

In a test case, test steps are classified into four groups:

- pre-condition
- action
- verification
- post-condition

Test steps belonging to pre-condition group run instructions necessary to set SUT to the required state before executing the main (action) step in the test case. The action step runs main instructions which are focus of the test case. Post condition steps are clean-up instructions for the test case (if necessary). Verification steps are assertions that match expected result to the actual result. For example, in Test case 1 steps 1-7 are preconditions for the action step 8, and step 9 and step 13 are verification steps. The test steps 14-17 are post-condition steps. It can be also noted that a test step can be pre-condition in one test case, post-condition in the second test case and action step in the third test case. Such flexibility facilitates re-usability of test steps in test cases.

TEST CASE 2: Attempting to create authorization without necessary consent level - verification of error message

1 Given that I logged in as ***"30029011111"***
2 Choose a representation ***"Myself"***
3 Visit a page that shows authorizations
4 Delete all given authorizations
5 Visit a page that shows settings
6 Set consent level to ***"@Level"***
7 Visit a page that shows authorizations
8 Create authorization with these parameters: ***"91028822222"***, ***"Lee"***, ***"@Area"***,***"2030-11-11"***
9 Verify message ***"Technical error is occured."*** reported

```
// Test parameters
```

Level	Area
2	10
1	10
1	9

The distinct feature of our test case design is that we separate assertions to its own test step as much as possible. In other words, we do not make assertions as a part of test step's implementation code but rather as a a separate test step. In fact, not only action step but pre-condition and post-condition steps have a related verification step as well. Ideally a test case shall have only one verification step i.e. a test case should focus testing on one specific feature of SUT. In fact, the action step can have several verification steps. We let FT to decide when or which assertions (i.e. verification test step) to use in a test case. For instance, action step 8 in Test case 1 and action step 8 in Test case 2 are the same. However, verification step 9 in these two test cases are completely different. Verification step 9 in Test case 1 verifies creation of authorization. Whereas verification step 9 in Test case 2 verifies the error message and authorization is not created.

Another feature of our test case design is that our test steps (in test case) are sequential and non-conditional. By non-conditional steps we mean executing an instruction on SUT without checking its state beforehand. Non-conditional test steps are simpler to comprehend. On the other hand, conditional test steps not only increase complexity of the test case but may also contribute to code duplication.

Our automated test cases can also be run manually when necessary. Test case are designed to be both machine interpretable (for automation) and human readable. Thus, we pay special attention to the readability, reusability and flexibility of test steps and test data. As can be seen from test case examples, test input is an integral part of the test step which makes automated test cases more readable (in terms of business process) and self-descriptive. All test inputs are specified within quotes ("") and we also try to apply styling (bold, italic) for easy comprehension. Furthermore, inputs can be specified both as a value and as a variable. For example, in step 8 in Test case 1 inputs "91028822222", "Lee" and "2030-11-11" are specified as the value while the input "@Area" as a variable. The values of the variable "@Area" are read from the parameter list. Depending on repeatability of the input, a tester can choose to set it as value or variable.

VI. Lessons learned and limitations

A. Benefits, lessons learned and recommendations

We have been using our test automation approach on Helsenorge since early 2017 almost without any change on architecture or significant re-implementation of already generated test artifacts unless it was required due to the changes in SUT. The main benefits we have achieved by applying automation are following:

1) *Saved manual effort, improved test coverage*.
By executing test cases automatically greatest benefits we achieved were, not surprisingly, saved manual testing effort and improved test coverage. Manual testing of one iteration takes about 5 minutes. Automated execution of all test iterations (more than 600) takes about 1 hour (over 50 times speedup). Automated test cases are part of Helsenorge's regression test suite which are run regularly, so automation benefits are not one time. Automated testing helped to increase not only breadth coverage but also depth coverage. The average depth coverage (ADC) was 5 which means a test case is executed with 5 different test inputs on average.
2) *Reduced maintenance cost*. We shifted (part of) intellectual work of creating automated test cases out of technical level (i.e. C# code) to the business level (test cases in Azure DevOps). In addition, not only maintenance of the test data (test inputs) but also verification of state (assertion) are made at the business level. All of these contributed to the lower maintenance costs. In addition, study reveals that merely automatically generating a set of test cases, even high coverage test cases, does not necessarily improve our ability to test software [6].
3) *Negative test steps*. An advantage of applying test automation at the API level is being able to create negative test cases by avoiding frontend validation. The negative test case verifies "fail-safe" (security) scenario of the application where a user intentionally or unintentionally attempts to perform prohibited action. The focus of the negative test cases is on verifying expected (error) messages such as "access denied", "technical error" etc. Test case 2 is an example of negative automated test case. As of today over 40% of our automated test cases are negative ones, see Table IV.

ICE and TCE metrics indicate how efficient test artifact from low level abstraction is transformed to higher level abstraction. ICE and TCE can also point out to cooperation between TD and FT. These metrics can be useful input on planning resources on similar automated projects, e.g. what is good proportion of FT to TD. We also believe that ICE, TCE and ADC metrics can be used as benchmarking criteria for comparing various automation solutions which are based on keyword- or process-driven technique. TCE has been previously defined by Gafurov et al. [9]. However, definition in formula 2 is more accurate compared to formula 1 in [9] because denominators in formulas 2 are from lower level of abstraction compared to numerators. In formula 1 in Gafurov et al. [9] denominator and numerator are at the same level of abstraction (i.e. both are at the test case level).

TABLE IV: Negative test cases

Team	Negative	Positive	All	Negative, %
POT	46	34	80	58
HOI	11	35	46	24
Total	55	79	126	44

Below is the list of the main recommendations based on our experience so far:

1) *Delegating*. Make test automation architecture, process and methods such that it is possible to delegate (at least) part of automation tasks to non-technical roles. This will contribute not only on cost reduction but also on involving more stakeholders in test automation. On the other hand, our own experience indicates that sometimes it can be challenging to hire skilled specialist to the role of TD and therefore delegating can help to address this challenge.
2) *Let TD have a FT role*. Starting to write test automation code without having holistic perspective on functionality results in inefficient test artifacts. This increases the risk of re-work and automation costs. This is especially true for junior TD or developers without testing mindset. Thus letting TD to have FT role when joining the team minimizes such risks and contributes on production of more robust test artifacts.
3) *V & V automated test case*. It is very important to thoroughly verify and validate (V & V) newly create automated test case and ensure that they work correctly. Failure or neglecting to quality assurance of automated test case may result on not trusting them by the FT and development team. While *testing* a test case, *test* not only success (happy path) but also make the test steps to fail and observe the error messages.
4) 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 of the test artifacts is essential. In fact, maintenance can be challenging and failure source of many test automation projects [11], [14]. Consequently, every test automation task shall be

evaluated 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. Another important point in this respect is that do not always focus on generating new artifacts without allocating time and resources for removing outdated ones. Not removing old artifacts on time will increase automation costs (e.g. new tester comes and spends time on them without knowing they are not actual anymore).

5) *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 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. In fact, prior to the developing our custom framework, we have been using SpecFlow [1] for automation. The drawback with the approach was that not only implementation code but automated test cases and test data were maintained in Visual Studio, and it was not fully integrated with our main test management system (i.e. Azure DevOps). In addition, it was challenging to organize test cases when their number increased. SUT with a large number of test cases require a proper test management system.

Although above mentioned aspects are important in test automation, they are not necessarily sufficient for success. Cultural aspect of test automation is probably even more influential factor. Characteristics such as engagement, commitment and interest in test automation are crucial success factors. Several practical recommendations are listed in Appendix A.

B. Limitations - opportunities for improvement

Several new features we wished we could have implemented within our automation framework. However, they were not implemented either because of the lack of resources or uncertainty around their ROI (return of investment).

The proposed metrics focus only on automation tasks related to "creating" new artifacts. Another type of automation task is "updating" efforts related to the existing artifacts which is not reflected in the proposed metrics. The challenge is automatic tracking of changes in two separate sources, namely test code in Visual Studio and test cases in Azure DevOps, and qualitatively unifying them. In general, this challenge belongs to the one identified in [2].

Our organization has also other products than Helsenorge where similar test automation technique is applied (e.g. VKP [5]). In addition, the same custom built automation framework is used. It can be interesting to estimate proposed test automation efficiency metrics on those products and compare with Helsenorge's one.

VII. Conclusion

In this work we summarize our test automation experience on large national e-health portal where automation has been applied for over 3 years (and still using). As of today, there are over 120 automated test cases which run as a part of our regression test suite. Each test case is executed on average with 5 different test inputs resulting in over 600 iterations. Out of 120 automated test cases over 40% of them are so called negative test cases which verify prohibited actions on system under the test. Such type of test cases is usually not easy to execute manually and can be easily overlooked. Our approach focuses on delegating automation tasks to non-technical tester to reduce costs and involve more stakeholders (with domain knowledge) into test automation. In addition, we shift assertion part of test automation from the technical level to the domain level. We proposed simple and complimentary automation metrics, such as implementation creation efficiency (ICE), test creation efficiency (TCE) and average depth coverage (ADC). The ICE and TCE metrics are useful not only to show automation status, but also can indicate cooperation between technical and non-technical testers. In addition, these metrics can be used as bench-marking criteria to compare various automation solutions. Last but not least, another contribution of the paper is to provide evidence of positive and real case experience of using test automation from long period of time to academic community.

VIII. Disclaimer

This paper represents the opinions of the authors to research community. It is not meant to represent the position or opinions of the authors' employer nor the official position of any staff members. Any errors are the fault of the authors.

References

[1] SpecFlow. https://specflow.org/.

[2] Nadia Alshahwan, Andrea Ciancone, Mark Harman, Yue Jia, Ke Mao, Alexandru Marginean, Alexander Mols, Hila Peleg, Federica Sarro, and Ilya Zorin. Some challenges for software testing research (invited talk paper). In *International Symposium on Software Testing and Analysis*, 2019.

[3] International Software Testing Qualifications Board. Test automation engineer – advanced level syllabus, version 2016.

[4] Direktoratet for e helse. Utviklingstrekk 2019 - beskrivelser av drivere og trender relevant for e-helse, 2019. Report on trend relevant for e-health. Report is in Norwegian.

[5] Direktoratet for e helse. Erfaringer og videre arbeid med velferdsteknologisk knutepunkt, 28.09.2018. Report is in Norwegian.

[6] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *International Symposium on Software Testing and Analysis*, 2013.

[7] Davrondzhon Gafurov. Applying model-based testing for new privacy and authorization concepts in Helsenorge. Technical report, Norsk Helsenett SF, 2019.

[8] Davrondzhon Gafurov, Margrete Sunde Grovan, and Arne Erik Hurum. Lightweight MBT testing for national e-health portal in Norway. In *ACM/IEEE International Conference on Automated Software Engineering*, 2020.

[9] Davrondzhon Gafurov, Arne Erik Hurum, and Martin Markman. Achieving test automation with testers without coding skills: an industrial report. In *ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[10] Helsenorge.no.

[11] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Software test automation in practice : Empirical observations. *Advances in Software Engineering - Special issue on software test automation*, 2010.

[12] Li Li and Wu Chou. Design and describe REST API without violating REST: A petri net based approach. In *IEEE International Conference on Web Services*, 2011.

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

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

APPENDIX A
TEST IMPLEMENTATION AND DESIGN GUIDELINES

1) *Test input*. Differentiate input data from body text of test step both for human readability and machine interpretation. Apply styling for easy readability if possible.
2) *Test input*. Enable flexible test input reading both as a value and as a variable. This helps to reduce updating efforts for tests with many iterations.
3) *Test step*. Make automated test steps as similar as possible to the manual test steps (e.g. use domain terms in test step definitions, avoid technical terms).
4) *Test step*. Make all important instructions implemented in the test code explicitly reflected in the test step definition text.
5) *Test step*. Make a test step independent from previous steps as much as possible. This will increase re-usability.
6) *Test step*. Avoid using conditional test steps.
7) *Assertion*. Make assertions a separate test step and not part of the action step i.e. make assertion part of the design not implementation.
8) *Test report*. Ensure test reports are generated automatically and presented in team's dashboard.
9) *Test report*. Show also historical results, not only current (last) one.
10) *Test report*. Ensure that the warning and error messages are understandable by non-technical tester. Prefer using domain terms than technical terms unless it is necessary.
11) *Cost*. Have a guideline on how automated test cases shall be coded, created, updated and eventually removed; Define criteria for selecting tests for automation.
12) *Cost*. Complex manual test cases not necessarily easy (or cheap) to automate.
13) *Cost*. Remember sometimes it can be cheaper to execute a test manually then to implement automated version.
14) *Cost*. Do not automate for automation sake.