From User Stories to End-to-end Web Testing

Humaid Mollah
Formal Methods and Tools Group
*University of Twente**
Enschede, The Netherlands
humaidalimollah73@gmail.com

Petra van den Bos
Formal Methods and Tools Group
University of Twente
Enschede, The Netherlands
p.vandenbos@utwente.nl

***Abstract*—In agile development of web applications, user stories are a common aid to develop the software. In this paper, we exploit this for deriving test cases from these user stories, to check and ensure the functioning of the application. Our step-by-step method is based on a literature review and interviews with professional testers. Besides steps for transforming user stories into test cases, the method also includes guidelines to select user stories beforehand, and guidelines on how to implement automatically executable test cases, afterwards. We demonstrate the method's applicability with case studies on three web applications from different application domains.**

***Index Terms*—user story, end-to-end test, web application, agile software development**

I. Introduction

Web applications take up a significant part of the software industry [20]. Consequently, efficient testing of such systems has become a major issue. Today, most software organizations follow an agile development methodology. This is an iterative approach for delivering projects to customers at a rapid pace [8]. With this paper we aim to find a testing approach that is suitable for web applications developed in such an agile environment. In particular, the testing approach should:

1) target web applications,
2) be a systematic approach targeting user-level tests,
3) fit into an agile development environment. In particular the approach should not introduce limiting restrictions or impose changes on the used development process, and
4) also require limited time for writing test cases, since web applications are developed at a rapid pace.

Using above requirements we motivate the approach of this paper. First, we note that most modern web applications (1) adopt a multi-tier architecture [17]. The application framework is then divided into multiple layers, like a database and a application server [9]. Also, a web application is commonly written in multiple programming languages. The challenges of combining components of such an architecture without making mistakes, make it valuable to test systematically at a user-level (2). In this paper, we therefore target end-to-end tests: tests that test the user workflow of an application from start to end.

Agile development (3) commonly includes the use of user stories to describe customer requirements [7] about the functionality of the web application [8]. Consequently, user stories give us a user-level description of system behaviour that is important to test. Using these user stories to formulate test cases therefore fits perfectly in an agile development environment. Moreover, no extra time (4) is required for obtaining user stories, as writing them is already part of the agile development process [8]. In this paper, we therefore investigate how to derive end-to-end test cases from user stories.

Though user stories and end-to-end tests are both defined at system level, translation from one to the other is non-trivial. A user story is only a sentence, usually in the format "As a ⟨user⟩ I want to ⟨goal⟩ so that ⟨reason⟩" [8]. An end-to-end test is an executable piece of code, and hence is more detailed. With a systematic approach, end-to-end test cases can be developed as soon as user stories have been defined, in parallel with the implementation of the user story in the web application itself, so that customer requirements are verified during the entire development process.

There exist fully automated approaches where a user story is automatically translated into test cases [1], [12], [19]. However, in those approaches, a template or a meta model has to be created for every user story by following a strict set of rules and keywords. These models have to be provided with all technical information about the parts of the web application, such as identifiers of input fields, buttons, or pages, or HTTP status codes. So though automated, these approaches require additional modelling effort, and information for creating executable test cases (similar information as for writing test cases manually). Moreover, a tester needs to go through a learning curve to learn how to do the modelling, adhere to the rules etc. In a fast-paced agile environment (3) this time-investment (4) may be too much compared to the development time for the web application. Instead we propose a more light-weight and systematic method that involves manually writing test cases.

We base our method, for developing test cases from user stories, on a combination of approaches from literature (Section II). Besides giving a comprehensive overview from literature on how to transform user stories into test cases, we also consider what happens before and after: how to select a user story, and how to implement a test case. Furthermore, we include interviews with professional testers to see what is done in practice (Section III). In Section V we show the applicability of our method by implementing test cases for three case studies from different application domains. The first case study is a web application for scheduling an event,

*This paper is based on the thesis of Humaid Mollah, which he wrote as part of his Computer Science bachelor at the University of Twente, and performed as an assignment for the company El Nino, located in Enschede.

the second for monitoring sensor devices, and the third for searching contact details of university employees. Section VI concludes and mentions future work. Additionally, this paper is accompanied by an artefact [3], containing the interview transcripts and the implemented test cases for our case studies.

The novelty of this paper is that we formulate a step-by-step method (Section IV) to test web applications from user stories, based on the combination and integration of all mentioned contributions: guidelines on how to select user stories, steps for transforming user stories into test cases, guidelines on how to implement test cases, and key points identified from the interviews with professional testers. Moreover, we apply this method on case studies. To the best of our knowledge, existing work only focuses on a subset of these elements. For example, [26] provide a survey of best practices for developing and implementing end-to-end tests (as in our guidelines), and [1], [12], [19] focus on user story to test case transformation (as in our literature review).

We target web applications for applying our method, by specifically giving guidelines for implementing test cases for web applications. However, we note that the first two steps of our method, where we select user stories, and transform user stories into test cases, are generic. Consequently, these two steps can also be applied on other types of applications.

II. Literature Review

Before discussing out literature review in the next sections, we first introduce the necessary terminology. First of all, a *test case* is a scenario in the web application. This includes any data inputs necessary to execute the scenario, and the expected result of this scenario. Test cases help determine whether the software satisfies a particular requirement [1], [15].

Secondly, a *workflow* is a series of processes or actions that the user performs from initiation to completion. An example of a workflow could be searching for a hotel followed by booking the hotel. Another example is creating an account (signing up) followed by logging into your account and then performing an action in your account. *End-to-end tests* execute user workflows in the web application [13].

The outline of next sections is as follows. Subsection II-A discusses the best practices for writing end-to-end tests for a web application such that the most important aspects are tested. The generation of end-to-end test cases from user stories will be discussed in Subsection II-B. This section presents a three-step procedure for identifying workflows, creating use-case scenarios, and defining test cases, from user stories. Finally, in Subsection II-C, discusses existing methods and tools for implementation of these end-to-end test cases.

A. Best practices for end-to-end testing

Many articles discuss the best practices for writing end-to-end tests [2], [4], [8], [13], [26], [27] Below we summarize the best practices applicable to our approach:

1) *Focus on the product's most important workflow.* The most important workflows of the web application should be tested first. For example, for booking a hotel, the most important workflow is searching locations and making a reservation. Therefore, this workflow and all user stories related to this workflow should be tested first.
2) *Large workflows should be broken down.* Large workflows in the application should be broken down into smaller workflows and actions, until the whole workflow can be expressed in terms of single user actions. This is because it is hard to find and trace back errors in a large workflow that is implemented as one large chunk of test code, compared to a workflow that is implemented as a sequence of small actions.
3) *Avoid low-level test cases.* When conducting end-to-end testing, it is important to only include test cases that are relevant for the user story being tested. Low-level test cases, which focus on verifying the functionality of small, individual components or modules of a software system, should be avoided. These tests typically verify specific functionality of a single unit of code, such as a function or method, and are also known as unit tests. For instance, when testing payment processing through a bank card, it is unnecessary to confirm whether charges are not applied on an expired VISA card, if a unit test for expired cards has already been conducted.
4) *Build test cases for all possible workflows.* A product must be tested for all possible interactions and micro-interactions that a user might have with the product. Therefore, the product should be tested for as many workflows as possible. Subsection II-B discusses how test cases can be formulated for these workflows.

Template	**Example**
As a ⟨type of user⟩	As a user
I want ⟨some goal⟩	I want to login to the webapp
so that ⟨reason⟩ (optional)	so that I can create a blog

Table I: Template of a basic User story.

B. Generation of test cases from user stories

In agile software development, user stories define software requirements [8]. They demonstrate a sequence of actions performed by the system to provide an observable result of value to the user [19]. User stories are not written in much detail [1], [8]. However, they tell the customer what to expect, a developer what to code, and a tester what to test [1]. Table I shows a template of a basic user story along with an example.

Various papers present different techniques and models to generate test cases from use cases and user stories [1], [8], [12], [14], [15], [19]. The application of user stories for the development of end-to-end acceptance tests has been discussed extensively in the book written by Cohn [8]. The conversion of use case scenarios into test cases has been studied comprehensively by Heumann [15]. There are also automated model-driven approach for the generation of test cases from use cases [1], [19]. Based on these papers, we define a three-step process to identify end-to-end test cases for a web application from its user stories. We will explain the steps by referring to the *User creates a Blog* example of

Scenario name	User Actions	Test inputs			Expected result
		Partition testing, User-based testing			
		user id	password	blog text field	
User creates blog successfully	Login, Create blog, Submit blog	valid	valid	non-empty	Display blog text
Unidentified user	Login	invalid	valid	-	Display login error
Unidentified user	Login	valid	invalid	-	Display login error
Empty blog	Login, Create blog, Submit blog	valid	valid	empty	Display alert about empty field
Server error	Login	any	any	-	Display server error

Table II: Test case definition for *User creates blog* example. With '-' we denote that no value is needed for the scenario.

Scenario Name	**Starting Flow**	**Alternate**
User successfully creates a blog	Basic Flow	
Unidentified user	Basic Flow	1
Empty blog	Basic Flow	2
Server error	Basic Flow	3

Table III: Use-Case Scenarios for *User creates blog* example.

Table I. The steps are as follows: Identify Flow (1), Develop Use-Case Scenarios (2), and Define Test Cases (3). We discuss each step in the next subsections.

1) Identify flow: The first step to generate test cases from user stories is to identify the flow of events in a user story. Such flows consists of *basic flows* and *alternate flows* [15]. The basic flow covers the flow of events that happen "normally". The alternate flow consists of events that refer to the "optional" or "exceptional" behavior of that user story. Alternate flows can be thought of as alternative routes to the basic flow. There can be more than one alternate flows for a user story.

From the example of Table I, we derive the following flows:

Basic Flow

1) Login : User accesses the web-app, system asks for user ID and password, user is authenticated.
2) Create a Blog: The system displays the homepage, the user clicks on the button "Create a Blog", and the system displays the text field of the Blog form.
3) Submit Blog: The user writes text in the blog form, the user presses "Submit Blog", and the system displays the created blog on the user's homepage.

Alternate Flow 1 Unidentified user : Invalid user ID or password entered for Login.

Alternate Flow 2 Empty Blog : Blog text-field is empty.

Alternate Flow 3 Server error : The server is not running.

2) Develop use-case scenarios: The second step is to define use-case scenarios for the workflows identified from the user story. *Use-case scenarios* define a complete path for that user story [1], [8], [15], [19]. Such a path can include, for example, just the basic flow, the basic flow plus alternative flow 1, or basic flow plus alternative flow 1 and 2 [15]. In theory, many combinations of flows are possible. However, the most important ones should be chosen following best practice (1) of Subsection II-A. Table III shows some of the important use-case scenarios for the *User creates blog* example.

3) Define test cases: The final step is to formulate test cases based on the use-case scenarios we have developed in the previous section. To do this, a table can be used as shown in Table II. This table is the test case definition with the actions of the use-case scenarios, the expected results and selected data inputs. Each row of the table represents one test case. Each scenario has a minimum of 1 test case but there can be more. Take the following three steps to define the test cases:

1) *Derive user actions* The use-case scenarios describe the combinations of basic and alternate flows, and the description of each of the flows provides the user actions of the flow. Use this to derive the user actions of each use case scenario.
2) *Identify data inputs*: Identify the different data fields needed for the user actions, and choose test inputs, i.e. values, for these data fields. For the *User creates blog* example, three fields can be identified from the user actions: student ID, password, and blog text field. The following strategies can be used for choosing test inputs:
   a) *Boundary testing* : Test inputs are chosen using boundary values [23]. For example, if the passing marks for an examination are 50%, the boundary values to test would be 49% and 50%. In this manner, both valid and invalid boundaries are tested.
   b) *Random testing* : Test inputs are chosen at random from the data domain. [11]. For example, we insert some random text in a text field, or choose a random number of products to buy.
   c) *Partition testing*: The set of inputs can be divided or partitioned into separate domains according to a self-defined rule. A test inputs is then chosen from each of these subdomains [29]. For example, inputs for a data field can be divided into a set of valid dates, e.g. 02-03-2004, and invalid dates, e.g. 02-30-2004.
   d) *Usage-based testing* : Select test inputs based on usage of the web application [25], e.g. by logging inputs used by users. For example, often a user chooses to buy both a phone and a phone sleeve within the same order.
3) *Identify expected results*: Identify the expected result of each test case. The expected result describes the state of the system after executing the user actions with its test inputs. For example, scenario *Unidentified student* (see Table II) has *Login error* as expected result for both tests.

InTable II, the test inputs are chosen using the partition testing approach, i.e., in Table II we have the partitions of valid and invalid user ids and passwords, and empty and non-empty blog fields. To choose a particular value from the partition, usage-based testing can be applied additionally. For the *Server error* scenario, any test input can be chosen because the expected result should always be a server error.

C. *Implementation of end-to-end tests*

When test cases have been defined, they still need to be implemented such that they can be executed. Literature suggests various techniques for the automation of end-to-end web testing [17]. These techniques can be broadly divided into two categories namely *Capture Replay Web Testing*, and *Programmable Web Testing*. The former refers to recording actions performed by a user on the web app (capture) and automatically executing the same actions (replay) which repeat the mouse movements and key-presses performed by the user [16], [17]. The latter uses test scripts to simulate the actions performed by a user with the help of specific testing frameworks [17]. The former is not a preferred option in our case for three reasons. Firstly, capture replay web testing is difficult to maintain and is not reusable [16]. Secondly, we cannot perform random testing by using this technique. Lastly, this technique cannot be used to test invisible web elements. Therefore, we will use programmable web testing, for the implementation and automatic execution of end-to-end test cases. Programmable web testing makes use of web elements such as input fields, links, buttons, etc. for test case execution. There are three methods to localize these web elements:

DOM-based Locate web page elements using information contained in the Document-Object-Model (DOM). A DOM is a programming interface for documents used on the web. It allows programs to change their structure, style, and content [6]. By using this approach, we can locate an element by its tag name or an attribute by its ID. This technique requires good development practices, and specifically good naming conventions, for writing code for the client-side (front-end), so that web elements can be identified by a unique identifier.

Coordinate-based Locate web elements by recording coordinates of a web page. This technique produces very fragile test scripts and is therefore considered obsolete [17].

Visual-based Locate web elements using image recognition to control GUI components. This technique requires the tester to make images of web elements so that parts of the web page can be located by checking for similarity with the web element. This can be a very lengthy process.

Literature suggests that the programmable DOM-based approach is the best option for implementing end-to-end tests [17], [18], [21], [26]. Firstly, DOM-based test cases do not require much time to develop. Secondly, the evolution of test cases can be done quickly and without much effort. Here, the word ‘evolution’ refers to updating test cases when application requirements and functionality change. Such changes are common in an agile development environment. Thirdly, DOM-based locators are proven to be more robust than visual locators. Lastly, tools for DOM-based approaches are readily available, e.g. Selenium Web Driver and Cypress.

III. Interviews

To gain comprehensive insights into end-to-end test case generation and implementation, we interviewed senior software testers from the web application development agency El Nino. These testers were carefully chosen based on their extensive experience in testing web applications using industry-standard tools such as Selenium and Cypress. As a result, their perspectives and experience with significant practical applications of end-to-end testing strategies provides valuable insights. The testers were interviewed individually and asked a set of pre-prepared questions aimed at eliciting their feedback on our literature review. Specifically, we asked them to give their insights on how to:

- Determine important functionality to test
- Develop test case scenarios from user stories
- Implement end-to-end test cases

In next three subsections, we summarize the information acquired in these interviews. In the last subsection, we report the key points which have been used to develop our method of Section IV. For interview transcripts, see the artefact [3].

A. *Determine important functionality to test*

The respondents mentioned that the important functionality of a web app can be identified from the product backlog. This is where the most important user stories for the current iteration of the project are marked. For example, in a priority queue in GitLab, the most critical functionalities are marked. However, they are not always sorted, so there can be multiple important user stories. In an agile working environment, the product owner is responsible for prioritizing user stories that need to be tested, and selects the user stories to test. Regarding the test cases that should be omitted, the testers agreed that low-level test cases can be avoided while writing end-to-end test cases. One of the testers said: “Indeed, user stories which are tested in unit tests can be skipped.”. These user stories are related to small bug fixes or features that may have a very low impact on the project.

B. *Develop test cases scenarios from user stories*

The respondents could relate the concepts of basic and alternate flows to their own techniques for identifying test cases. One of the testers said, “I call the basic flow the happy path”. Another tester explained, “alternative flows are flows that are likely to be used apart from the regular flow”. The respondents mentioned that the number of alternate flows depends on the functionality being tested. Since a lot of the alternate flows are covered via unit tests, the end-to-end test cases should be used to test broader “exceptional” scenarios where both the front-end and back-end of the web application are used. The selection of alternate flows also depends on the “likelihood of breaking” as mentioned by one of the respondents. Moreover, one tester said that the “80/20 principle usually does apply”. This was a reference to the Pareto principle which states that 80% of the consequences (alternate flows) originate from 20% of the causes (parts of code or functionalities) [10]. About choosing test inputs, the testers noted that boundary testing was a good approach to test both positive and negative scenarios. One of the testers said that “usage-based testing could be considered as the most commonly used option”.

C. *Implement end-to-end test cases*

All the testers agreed that Selenium and Cypress are the best approaches for implementing end-to-end test cases. One of the testers said, “these are the easiest options to implement end-to-end tests”. The other options, such as visual-based locators, are not particularly applicable for testing big web applications where many different scenarios have to be considered.

D. *Interview results*

The key points that were identified in the interviews are:

1) The most important functionality of the web application is subject to the current iteration in the development of the project. Important functionality can be found in the product backlog where the most important user stories are marked with tags or are in the priority queue.
2) User stories are prioritized by the product owner, who should always be consulted if it is not clear which functionality is most important.
3) Low-level test cases, for testing e.g., small bug fixes, or small feature updates with low impact, can be omitted.
4) The number of alternate flows depends on the functionality being tested. End-to-end test cases should target broader scenarios involving both client and server side.
5) The 80/20 Pareto rule [10] applies on alternate flows, i.e., most or 80% of alternate flows are caused by a small part or 20% of the code, or a functionality.
6) Data inputs for test cases are commonly chosen using usage-based testing and boundary testing.

IV. Method

From the literature review and interviews we now define our step-by-step method to derive end-to-end test cases from user stories. We describe three top-level steps: Select User Story (A), Transform User Story into Test Cases (B), and Implement Test Cases (C), which are each refined into smaller sub-steps.

A. *Select user story*

The first step in the method is to select the user stories for which to develop end-to-end test cases. From the best practices for end-to-end testing (Subsection II-A) and the interviews results (Section III), we identified the following guidelines that should be taken into account for this selection:

1) *Identify important functionality* : The user story which defines the most important functionality of the web application should be chosen first (see Subsection II-A (1)). In an agile development environment, the Product Owner is responsible for defining user stories and prioritizing important functionality for each iteration. He either marked user stories with priorities, or can be asked to point out the most important user stories that should be tested (see Subsection III-D(1) and (2)).
2) *Omit low-level user stories*: User stories that define requirements that are covered by unit tests should be omitted (see Subsection II-A (3)). These user stories can be related to small bug fixes or a small feature update that does not have much impact on the web app (see Subsection III-D(3)).

B. *Transform user story into test cases*

The second step in the method is to transform the user story into end-to-end test cases. To do this we follow the three steps we identified in our literature review (Subsection II-B).

1) *Identify flow*: First identify the basic flow of the user story. This basic flow can usually be divided into user actions (see Subsection II-A (2)). Additionally, identify the alternate flows of the user story. Alternate flows can be found by considering alternatives for the user action from the basic flow. It is important to note that there can be many alternate flows related to one user action as compared to the others (see Subsection III-D (4) and (5)). We note that identifying basic and alternate flows is a creative process. However, to test an application thoroughly, one should aim at finding all workflows that a user could use (Subsection II-A (4)).
2) *Develop use-case scenarios*: Based on the basic and alternate workflows, determine the different paths that could be followed by the user. A table like Table III can be used to keep track of each scenario. Use case scenarios should be worked out for complex cases involving multiple alternate flows, if possible, to increase effectiveness of testing (Subsection II-A (4)).
3) *Define test cases*: Identify the user actions, data inputs, and expected results for each use-case scenario, as in subsubsection II-B3. Data inputs can be chosen using one of the standard data selection methods from subsubsection II-B3: boundary value, random, partition, or usage-based testing. We note that while the interviewed testers indicated that usage-based and boundary value testing occurs most, choosing one of these two methods does not *always* make sense, and depends highly on the data domain. For example, if a text field allows any text, there are no boundaries. Furthermore, it is probably insufficient to only test passwords that have been used before.

C. *Implement test cases*

The test cases defined in the previous step can be implemented using a programmable DOM-based approach using tools such as the Selenium Web Driver or Cypress (see Subsection II-C). We structure the implementation of test case code, according to the test definitions, as follows:

1) *Implement user-actions*: Implement each user action with the help of DOM-based commands to execute the action in the web application. Each user action can be implemented with a method definition. If the user action requires use of test inputs, then these test inputs are arguments of the method. If a user action requires multiple clicks, and/or filling in multiple fields, etc., then we split the user action in multiple implementation actions and define them with individual method definitions (for ease). For example, *submitting a blog* consists of (1) typing the text in the form *and* (2) pressing the “Submit blog”

Scenario name	User Actions	Test inputs						Expected result
		Random testing, Partition testing, Usage-based testing						
		Email	Password	Date	Times	Client email	Session form	
Event scheduled successfully	Login, Create, Submit	valid	valid	valid	valid	valid	non-empty	Success notification
Unidentified professional	Login	random	random	-	-	-	-	Error notification
Invalid date-field	Login, Create, Submit	valid	valid	invalid	valid	valid	-	Error notification
Invalid time-frame	Login, Create, Submit	valid	valid	valid	invalid	valid	non-empty	Error notification
No clients chosen	Login, Create, Submit	valid	valid	valid	valid	empty	non-empty	Error notification
Invalid client email	Login, Create, Submit	valid	valid	valid	valid	invalid	-	Cannot click save

Table IV: Test case definitions for MyDay case study.

button. These can be treated as 2 implementation actions of a single user action "Submit Blog".

2) *Implement test case definitions*: Implement each test case definition, i.e. each row of the table, with a method that invokes the user actions defined earlier. Sleep/wait commands should used between user actions to account for the time taken for a page or DOM elements on a page to load. Concrete input values are passed via the arguments of the user actions. These inputs are chosen according to the strategies chosen in the test case definition. Some creativity of the tester may be applied here to choose 'interesting' inputs. After calling the methods of the user actions, the test case method asserts the expected result. For example, by asserting that the message of a notification contains a word like "Success". This way, any unexpected result of executing this test case will be reported.
3) *Execute all test cases automatically*: Implement a main method calling the methods of all test case definitions.

V. Case studies

This section analyzes the applicability of the method of Section IV by doing a case study for a sample of three deployed real-world web applications: MyDay, VRM, and UTwente people page. The MyDay application is a scheduling application for appointments. The VRM application is a monitoring application for sensor data from field devices. The UTwente people page enables searching for UTwente employees. These web applications belong to different application domains.

In the following subsections, we discuss each case study. First, we provide a more detailed description of what the web application does, the technologies it uses, and the selected user story. After that, we transform the user story in test cases, as we described in our literature review: we list the basic and alternate flows, work out the use case scenarios, and define the test cases. Finally, we provide some details about the implementation of the end-to-end test cases and give the results of executing the test cases.

For each case study, we implement test cases using Python's built-in unit testing framework, and the Selenium Web driver [28]. We chose to use Selenium's web driver for Chrome, but this can be easily adapted to use other browsers like Firefox, or Safari. All test cases have been developed by following the best practices for using the Selenium web driver [24]. For example, the Page-Object-Model (POM) has been used. POM is a design pattern in Selenium that stores all the web elements in an object repository. Moreover, if IDs have been used in HTML tags, they are used as the primary web element locators. Only if IDs are not provided, other locators such as the CSS class name and XPath locators are used. We provide the code of the implemented test cases in our artefact [3].

Scenario Name	**Starting Flow**	**Alternate**
Event scheduled successfully	Basic Flow	
Unidentified professional	Basic Flow	1
Invalid date-field	Basic Flow	2
Invalid time-frame	Basic Flow	3
No clients chosen	Basic Flow	4
Invalid client email	Basic Flow	5

Table V: Use case scenarios for MyDay case study.

A. *Case 1: MyDay: scheduling web application*

MyDay is an appointment scheduling application which connects sports professionals with their clients. The web application API is built mainly using PHP and the front-end of the application is built with Vue which is a JavaScript framework. We choose the following user story for this case study, as it represents the main workflow of the application:

As a professional, I want to schedule a session with a client.

1) Transform user story into test cases:

Basic Flow

1) Login: The professional accesses the MyDay web app, the professional clicks "Login", the system asks for email and password, the professional is authenticated and redirected to the MyDay Professional homepage.
2) Create event: Professional clicks "Create Event", the system displays the drop-down menu, professional clicks "Create an event with client", and the system displays session form.
3) Submit event: Professional fills session form, professional clicks "Save", the system notifies the professional that the event has been created succesfully.

Alternate Flow 1 *Unidentified professional* : Invalid user email or password entered for Login.

Alternate Flow 2 *Invalid date field* : Invalid (past) date selected for scheduling an event.

Alternate Flow 3 *Invalid time-frame* : Invalid time period selected for scheduling an event.

Alternate Flow 4 *No clients chosen* : No clients are selected for creating an event.

Scenario Name	User Actions	Test Inputs			Expected Result
		Partition testing, Random testing			
		Email	Password	Device id	
User can monitor device	Login, Navigate, Monitor	valid	valid	random or as in user story	Display data
Unidentified user	Login	invalid	invalid	-	Login error
No access to device	Login, Navigate	valid	valid	random	Error notification
Device error	Login, Navigate	valid	valid	random	Error notification

Table VI: Test case definitions for VRM case study.

Alternate Flow 5 *Invalid client email* : An invalid email is entered to create an event.

The use case scenarios are shown in Table V.

For choosing the test inputs for test cases, we use random testing and partition testing. A randomly generated, invalid input for the email and password fields should produce a login error and a randomly generated input for the client email should alert the user. Furthermore, an invalid (past) date-field or an invalid time frame should alert the user while a valid date and time should produce no errors. Also, an empty session form should alert the user but a non-empty form should produce no errors. Table IV shows all test case definitions.

2) Implementation of end-to-end test cases: Next, we give some implementation details. For the "Login" user action, we locate the username and password fields with the help of "id" tags and type the text in these fields by using selenium's built in functions. The "Submit event" user action is broken down into various implementation actions such as "choose date", "choose time" and "enter email", which are each implemented with a method definition, with arguments for the test inputs. For the *Event scheduled successfully* test case, the expected result is checked by asserting that the raised notification has a "Success" message. Similarly, for the other test cases, an "Error" notification is asserted. For the test case *Invalid client email*, we assert that the "Save" button cannot be clicked. This is done by catching the exception from the click method of this button.

3) Results: Four out of six test cases did not produce the expected results. These test cases were related to the following alternate Flows : *Invalid date-field* and *Invalid client email*. Two bugs were discovered:

- a professional can create an event with an invalid client email, and
- a professional can submit an event with an invalid date.

B. Case 2: VRM: remote monitoring web application

VRM is a remote monitoring web application that allows users to remotely control energy systems such as freezers, solar chargers, and water tanks. The application performs real-time data collection on the devices and displays this to the user. The application API is built using PHP. The front-end of the application is built with JavaScript, Vue, and Less. Since the sole purpose of this application is to monitor devices in real-time, we select the following important user stories:

As a VRM user, I want to monitor:

1) *the time of the device that was updated the latest*
2) *the temperature of the water tank and the freezer,*
3) *the water level of the water tank, and*
4) *my device's alarms.*

Scenario Name	**Starting Flow**	**Alternate**
User can monitor device	Basic Flow	
Unidentified user	Basic Flow	1
No access to device	Basic Flow	2
Device error	Basic Flow	3

Table VII: Use case scenarios for VRM case study.

1) Transform user story into test cases: In this case study, we formulate the actions of the basic and alternate flows for any device. The device id is then a test input for the test cases.

Basic Flow

1) Login: User navigates to the VRM web application and logs in with his email and password. System displays the homepage.
2) Navigate to device: User navigates to the device list, the system displays all the installed devices. The user selects a device to monitor by the "id".
3) User monitors device: The user monitors a device by clicking the device name, the system displays the relevant device information.

Alternate Flow 1 *Unidentified user* : Invalid user email or password entered for Login.

Alternate Flow 2 *No access to device* : The device is not accessible to this user.

Alternate Flow 3 *Device error* : Device not connected or data cannot be read from the device.

The use case scenarios of the VRM case study are shown in Table VII, and the test cases definitions in Table VI. Like for MyDay, test inputs for email and password are chosen using random and partition testing. In the test case *User can monitor device* we test the devices as far as this is specified by the user story: we check the general field displaying the time (1), the temperature of devices water tank and freezer (2), the level of device water tank (3), and the alarms of a random device (4). In other test cases, a random device id is used as test input as well.

2) Implementation of end-to-end test cases: In this case study, the 'Login' user action is implemented similarly as for *MyDay*. The 'Navigate' user action is broken down 2 implementation actions namely "navigate to device list" and "get device by id". Furthermore, the "Monitor" user action is also divided into several implementation actions such as "check last updated" and "check water tank level". The expected result of displaying data is implemented by asserting that the required

Scenario name	User Actions	Test inputs		Expected result
		Partition testing, Usage-based testing		
		Query	Filter	
User can search for contact details	Search,Filter,Click	UT employee	none	Display contact details
Unidentified employee	Search	non-UT employee	none	No results
Non-matching filter	Search,Filter	UT employee	too restrictive filter	No results

Table VIII: Test case definitions for UTwente people page case study.

information is really present, i.e. the respective DOM elements needs to be not "None" or "null". For test cases *No access to device* and *Device error*, we assert that a notification with an "Error" message is raised. We note that the *Device error* test case is only executable when we have a device which is not connected to the hardware.

3) Results: All test cases produced the expected results.

C. Case 3: UTwente people page: searching web application

The University of Twente people page is a searching web application used by university students and employees to search for information about university employees. The website is built using PHP and several external libraries to search the university database. Considering the main use case is searching, we select the following user story:

As a student, I want to search for the contact details of a UTwente employee

1) Transform user story into test cases:

Basic Flow

1) Search: User navigates to the UTwente people page and makes a query in the search bar. System displays the results.
2) Filter: User filters the results by choosing one of the filters. The system refines the result.
3) Click to see contact details: User clicks on one of the results, and then on the "Contact" button, to see the contact details.

Alternate Flow 1 *Unidentified employee* : Query does not match any persons in the database.

Alternate Flow 2 *Non-matching filter* : The applied filter returns no results for the query.

The use case scenarios are shown in Table IX.

For the test case definitions, shown in Table VIII, we select test inputs using partition and usage-based testing. With partition testing, we differentiate between UT employees and non-UT employees. We choose a known UT employee (i.e. Petra van den Bos) and a known non-UT employee (i.e. Humaid Mollah) for the test cases: *User can search for contact details*, and *Unidentified employee*, respectively. The expected results are displaying the contact details of Petra and no results for searching Humaid. For the *Non-matching filter* test case, we choose the same known UT employee, but select a filter such that no results are displayed.

2) Implementation of end-to-end test cases: Each user action discussed in the basic flow is implemented with one method. The "search" user action is implemented by using selenium's built in functionality for typing text. The method for the "filter" user action takes one of the 3 filters as an argument, and locates and activates the respective filter using its class name tag. Like for the VRM case study, displaying contact details is checked by asserting that elements are not "None" or "null". To check for no result, we assert that clicking on a result element raises a "NoSuchElementException".

3) Results: All test cases produced the expected results.

Scenario Name	Starting Flow	Alternate
User can search for contact details	Basic Flow	
Unidentified employee	Basic Flow	1
Non-matching filter	Basic Flow	2

Table IX: Use case scenarios for UTwente people page case study.

VI. Conclusion

We have developed a user-story-driven approach for writing and implementing end-to-end test cases for web applications developed in an agile environment. We provide a three-step method for deriving executable end-to-end test case implementations from user stories. Our method is based on a literature review and interviews with testers. We demonstrated the method's applicability with three case studies.

In future work we would like to automate (parts of) our method. For example, given a test definition table (e.g., Table II), we could automatically generate a code template. Furthermore, we could provide tooling to support the tester taking all steps for transforming user stories into test case definitions. Moreover, we could use automatic approaches for choosing test inputs [22]. Additionally, we would like to perform more case studies to further improve our method and gain additional confidence for the applicability of our method. Lastly, we are interested in combining our work with model-based testing approaches, to increase the number and coverage of test cases [5].

References

[1] Allala, S.C., Sotomayor, J.P., Santiago, D., King, T.M., Clarke, P.J.: Towards transforming user requirements to test cases using MDE and NLP. In: 2019 IEEE 43rd COMPSAC. vol. 2, pp. 350–355 (2019), https://doi.org/10.1109/COMPSAC.2019.10231

[2] Apostolov, A., Vandiver, B.: End to end testing - What should you know? In: 2014 67th Annual Conference for Protective Relay Engineers. pp. 125–131 (2014), https://doi.org/10.1109/CPRE.2014.6798999

[3] Artefact of this paper, https://github.com/hnm27/userstories-to-testcases, Containing: executable case study test cases, and interview transcripts.

[4] Bai, X., Tsai, W., Paul, R., Shen, T., Li, B.: Distributed end-to-end testing management. In: Proc. of 5th IEEE EDOC. pp. 140–151 (2001), https://doi.org/10.1109/EDOC.2001.950430

[5] van den Bos, P., Tretmans, J.: Coverage-based testing with symbolic transition systems. In: Beyer, D., Keller, C. (eds.) TAP. pp. 64–82. Springer, Cham (2019), https://doi.org/10.1007/978-3-030-40914-2_10

[6] Brucker, A.D., Herzberg, M.: A formal model of the document object model. Archive of Formal Proofs (December 2018), https://isa-afp.org/entries/Core_DOM.html

[7] Chopade, M.R.M., Dhavase, N.S.: Agile software development: Positive and negative user stories. In: Proc. of 2nd I2CT. pp. 297–299 (2017), https://doi.org/10.1109/I2CT.2017.8226139

[8] Cohn, M.: User Stories Applied: For Agile Software Development. Addison-Wesley signature series, Addison-Wesley (2004), https://books.google.nl/books?id=SvIwuX4SVigC

[9] Diao, Y., Hellerstein, J., Parekh, S., Shaikh, H., Surendra, M.: Controlling quality of service in multi-tier web applications. In: Proc. of 26th IEEE ICDCS. pp. 25–25 (2006), https://doi.org/10.1109/ICDCS.2006.23

[10] Dunford, R., Su, Q., Tamang, E.: The pareto principle. The Plymouth Student Scientist **7**, 140–148 (2014)

[11] Godefroid, P.: Random testing for security: Blackbox vs. whitebox fuzzing. In: Proc. of the 2nd RT. p. 1. Assoc. for Computing Machinery, New York, NY, USA (2007), https://doi.org/10.1145/1292414.1292416

[12] Granda., M.F., Parra., O., Alba-Sarango., B.: Towards a model-driven testing framework for GUI test cases generation from user stories. In: Proceedings of the 16th ENASE. pp. 453–460. INSTICC, SciTePress (2021), https://doi.org/10.5220/0010499004530460

[13] Gundecha, U., Avasarala, S.: Selenium WebDriver 3 Practical Guide: End-to-end automation testing for web and mobile browsers with Selenium WebDriver, 2nd Edition. Packt Publishing (2018), https://books.google.nl/books?id=_AhnDwAAQBAJ

[14] Gutiérrez, J.J., Escalona, M.J., Mejías, M., Torres, J.: An approach to generate test cases from use cases. p. 113–114. ICWE '06, ACM, New York, NY, USA (2006), https://doi.org/10.1145/1145581.1145606

[15] Heumann, J.: Generating test cases from use cases. The rational edge **6**(01) (2001), https://www.researchgate.net/publication/220724437_Generating_Test_Cases_from_Sequences_of_Use_Cases

[16] Leotta, M., Clerissi, D., Ricca, F., Tonella, P.: Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In: Proc. of 20th WCRE. pp. 272–281 (2013), https://doi.org/10.1109/WCRE.2013.6671302

[17] Leotta, M., Clerissi, D., Ricca, F., Tonella, P.: Ch. 5 - approaches and tools for automated end-to-end web testing. Adv. in Comp., vol. 101, pp. 193–237. Elsevier (2016), https://doi.org/10.1016/bs.adcom.2015.11.007

[18] Ma, X., Wang, N., Xie, P., Zhou, J., Zhang, X., Fang, C.: An automated testing platform for mobile applications. In: 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). pp. 159–162 (2016), https://doi.org/10.1109/QRS-C.2016.25

[19] Massod, M., Iqbal, M., Khan, M., Azam, F.: Automated user story driven approach for web based functional testing. International Journal of Computer and Information Sciences **11** (06 2017)

[20] Offutt, J.: Quality attributes of web software applications. IEEE Software **19**(2), 25–32 (2002), https://doi.org/10.1109/52.991329

[21] Pelivani, E., Cico, B.: A comparative study of automation testing tools for web applications. In: 2021 10th Mediterranean Conference on Embedded Computing (MECO). pp. 1–6 (2021), https://doi.org/10.1109/MECO52532.2021.9460242

[22] Popić, S., Pavković, B., Velikić, I., Teslić, N.: Data generators: a short survey of techniques and use cases with focus on testing. In: 2019 IEEE 9th ICCE. pp. 189–194 (2019), https://doi.org/10.1109/ICCE-Berlin47944.2019.8966202

[23] Ramachandran, M.: Testing software components using boundary value analysis. In: 2003 Proceedings 29th Euromicro Conference. pp. 94–98 (2003), https://doi.org/10.1109/EURMIC.2003.1231572

[24] Ramya, P., Sindhura, V., Sagar, P.V.: Testing using selenium web driver. In: Proc. of 2nd ICECCT. pp. 1–7 (2017), https://doi.org/10.1109/ICECCT.2017.8117878

[25] Regnell, B., Runeson, P., Wohlin, C.: Towards integration of use case modelling and usage-based testing. Journal of Systems and Software **50**(2), 117–130 (2000), https://doi.org/10.1016/S0164-1212(99)00084-9

[26] Ricca, F., Stocco, A.: Web test automation: Insights from the grey literature. In: Bureš, T., Dondi, R., Gamper, J., Guerrini, G., Jurdziński, T., Pahl, C., Sikora, F., Wong, P.W. (eds.) SOFSEM 2021: Theory and Practice of Computer Science. pp. 472–485. Springer International Publishing, Cham (2021), https://doi.org/10.1007/978-3-030-67731-2_35

[27] Tsai, W., Bai, X., Paul, R., Shao, W., Agarwal, V.: End-to-end integration testing design. In: Proc. of 25th COMPSAC. pp. 166–171 (2001), https://doi.org/10.1109/CMPSAC.2001.960613

[28] Web driver Selenium, https://www.selenium.dev/documentation/webdriver/, retrieved January 18, 2023

[29] Weyuker, E.J., Jeng, B.: Analyzing partition testing strategies. IEEE Trans. Softw. Eng. **17**(7), 703–711 (jul 1991), https://doi.org/10.1109/32.83906