Test Driven Development of Web Applications: a Lightweight Approach

Diego Clerissi, Maurizio Leotta, Gianna Reggio, Filippo Ricca

Abstract:

The difficulty of creating a test suite before developing a web application is the main barrier to the adoption of the Acceptance Test Driven Development (ATDD) paradigm.

In this work, we present a general lightweight approach and a specific instantiation based on existing tools for acceptance test driven development of web applications. The idea, which is the basis of our approach, is simple: using a capture/replay tool able to generate test scripts on previously created Screen Mockups of the web application to develop. These test scripts can be later executed against the web application, and used to drive its development following the requirements in ATDD mode.

The presented approach has been used to re-develop the main features of an open-source web application. Observations, limitations of the approach, and lessons learnt are outlined.

Digital Object Identifier (DOI):

`http://dx.doi.org/10.1109/QUATIC.2016.014`

Test Driven Development of Web Applications: a Lightweight Approach

Diego Clerissi, Maurizio Leotta, Gianna Reggio, Filippo Ricca
Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
diego.clerissi@dibris.unige.it, maurizio.leotta@unige.it, gianna.reggio@unige.it, filippo.ricca@unige.it

***Abstract*—The difficulty of creating a test suite before developing a web application is the main barrier to the adoption of the Acceptance Test Driven Development (ATDD) paradigm.**

In this work, we present a general lightweight approach and a specific instantiation based on existing tools for acceptance test driven development of web applications. The idea, which is the basis of our approach, is simple: using a capture/replay tool able to generate test scripts on previously created Screen Mockups of the web application to develop. These test scripts can be later executed against the web application, and used to drive its development following the requirements in ATDD mode.

The presented approach has been used to re-develop the main features of an open-source web application. Observations, limitations of the approach, and lessons learnt are outlined.

***Index Terms*—ATDD paradigm, Screen Mockups, Capture/Replay tools**

I. Introduction

Developing modern web applications is a big challenge for software companies, because they undergo through ultra-rapid development cycles, due to customers' requests and requirements evolution, pushing new features and bug fixes to production within days. In this context, agile approaches and automated testing frameworks are considered among the best choices for web application development and quality assurance [15].

Acceptance Test Driven Development [5] (from now, ATDD) is a cornerstone practice that puts acceptance testing and refactoring on top of the software development process. In brief, ATDD is a development process based on short cycles: first, an initial set of failing test scripts is built starting from a feature's specification, usually expressed by means of a user story (i.e. a simple text description consisting of one or more short sentences); then, some code is written to pass each test, and finally, some refactorings are applied to improve the structural code quality of the web application. However, ATDD is not limited to agile contexts where requirements can be expressed with user stories. Indeed, even more formal requirements specifications based on use cases, such as the one proposed by Reggio et al. [19], [20], can be used with the ATDD paradigm. Usually, in these cases, acceptance tests are defined by following the scenarios composing the use cases.

Even if a large number of testing tools usable in the web applications context emerged in the last years (for instance, Fitnium[1], an integration of FitNesse[2], where test cases can be represented in a tabular form by using the natural language, and Selenium IDE[3], a Firefox plug-in that allows to record, edit, and execute web test scripts), it is well-known that the difficulty of creating a test suite, before the web application exists, prevents the usage of the ATDD paradigm in a real context [8]. Usually, developers wishing to apply ATDD must manually write complex executable test scripts [2]. This task is cumbersome due to the tight coupling between test scripts and web applications. In fact, in order to work, test scripts must be able to locate user interface elements (i.e. web elements) at run-time by using specific hooks (for example identifiers contained in web elements) and interact with them. Unfortunately, without having the web application it is difficult to foresee such hooks.

In order to simplify the adoption of the ATDD paradigm in the web context, we propose a general lightweight semi-automatic approach that, starting from textual requirements and screen mockups, is able to generate executable functional web test scripts (i.e. black box tests able to validate a web application by testing its functionalities), which in turn drive the development of a web application. Once created the web application through ATDD, the test scripts are refactored removing potential reasons of fragility (e.g. hooks that are likely to break during the web application evolution), clones, and extended by means of input generator tools to form a robust regression test suite. This will help developers to produce high quality web applications.

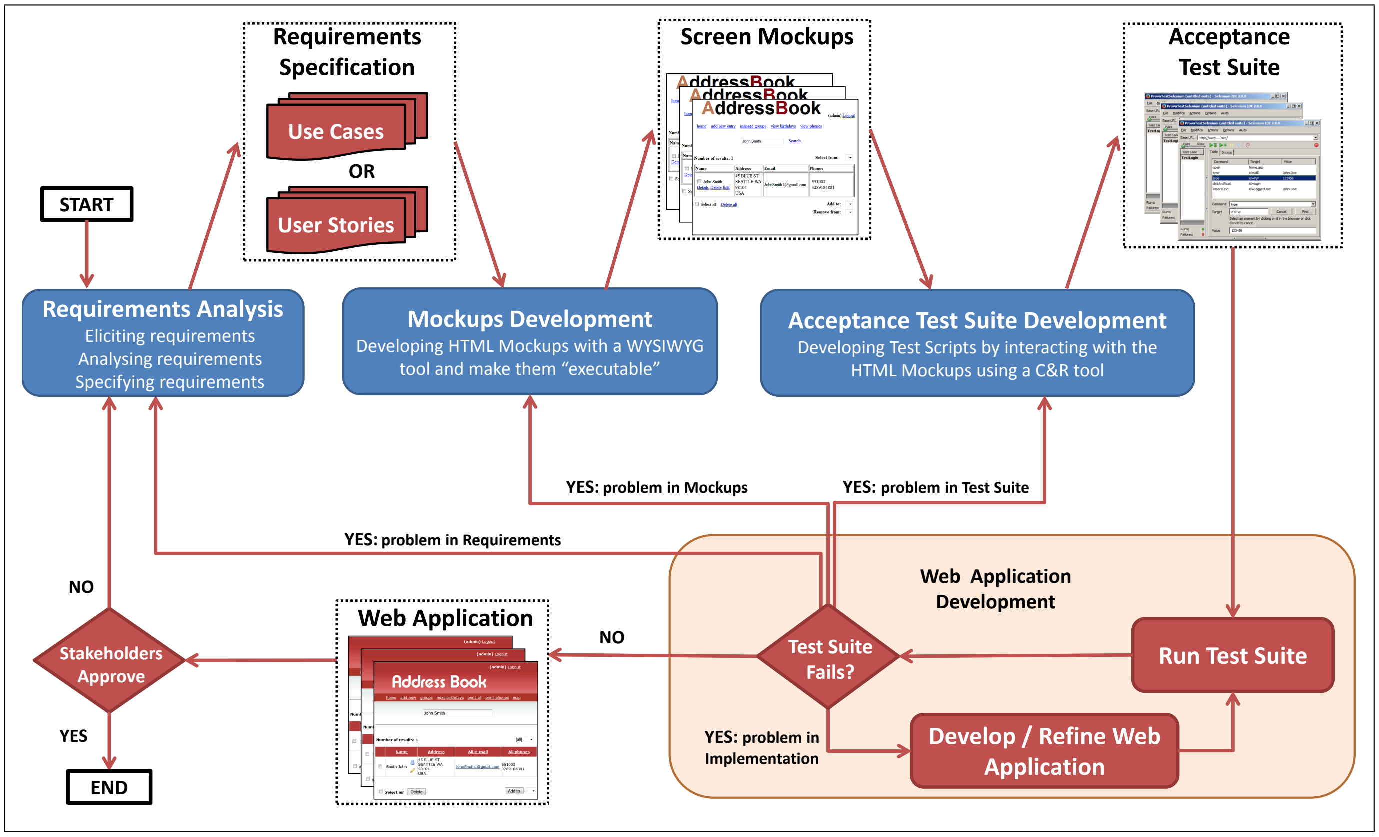
This paper extends and refines our previous poster paper [4], where we briefly sketched for the first time the idea. This paper is organized as follows: Section II describes our general approach for acceptance test driven development of web applications. Section III instantiates the approach with a set of selected open source tools, and reports some observations, limitations and lesson learnt resulting from its usage to develop the AddressBook web application. Finally, Section IV presents the related works, followed by conclusions and future work in Section V.

II. The Approach

In this section, we describe in detail the tasks composing the proposed approach, from those, shown in Figure 1, supporting the development of the web application, to the ones, shown in Figure 2, concerning the development of the regression test suite used during the evolution of the target web application.

[1]https://fitnium.wordpress.com/
[2]http://www.fitnesse.org/
[3]http://www.seleniumhq.org/projects/ide/

Fig. 1. From the Requirements Analysis to the Web Application Development


START
Requirements Specification
Use Cases
OR
User Stories
Screen Mockups
Acceptance Test Suite
Requirements Analysis
Eliciting requirements
Analysing requirements
Specifying requirements
Mockups Development
Developing HTML Mockups with a WYSIWYG tool and make them "executable"
Acceptance Test Suite Development
Developing Test Scripts by interacting with the HTML Mockups using a C&R tool
YES: problem in Mockups
YES: problem in Test Suite
YES: problem in Requirements
Web Application Development
NO
Web Application
NO
Stakeholders Approve
Test Suite Fails?
Run Test Suite
YES
END
YES: problem in Implementation
Develop / Refine Web Application

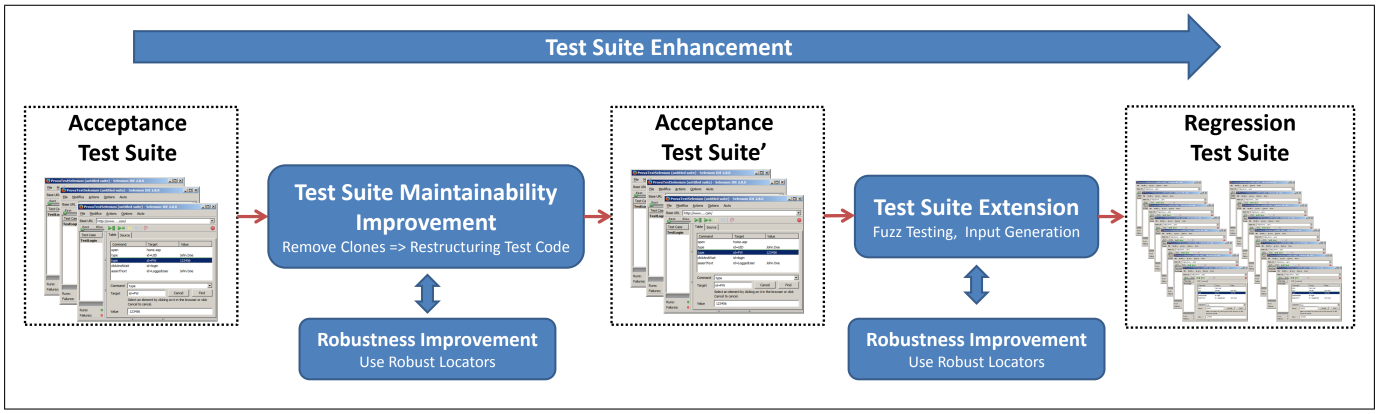
A. *Requirements Analysis*

Requirements analysis aims at producing the requirements specification for the web application under development. It includes the following subtasks: (1) eliciting the requirements from future users, customers and other stakeholders, (2) analysing the requirements to understand whether they are complete, consistent, and unambiguous, (3) specifying requirements as use cases or user stories depending on whether, respectively, a more prescriptive or more agile development approach is adopted.

B. *Mockups Development*

Mockups development aims at creating a set of screen mockups used for prototyping the user interface of the web application to develop [7], [17]. In order to reduce as much as possible the need of manual intervention required to run the automated acceptance test scripts on the web application under development, the mockups have to represent quite accurately – from a functional point of view – the interfaces of the web application (i.e. all the web elements of the web pages to interact with must be shown in the mockups, while the layout

Fig. 2. Regression Test Suite Development



and the styles can be just sketched). Since our goal is to use a capture/replay tool able to generate test scripts on previously created screen mockups of the web application to develop, a WYSIWYG content editor that creates HTML pages represents the best choice to quickly develop them. The WYSIWYG content editor could be also used to specify the properties of the web locators (i.e. hooks that point to the web elements to interact with). Locators can use many different properties; the most reliable ones are identifiers, linkText, and name values [14].

C. Acceptance Test Suite Development

Once the mockups are available, it is possible to record the test suite by interacting with them. To make this task easier and to simulate screen mockups navigability, we suggest to implement: (1) the links among the mockups and (2) the submission buttons. Concerning submission buttons, it is possible to hard-code the alternative links to different target mockups using JavaScript; for instance, when dealing with a login form we can reach two mockups, "homePage.html" and "wrongPage.html", depending on the inserted values. In this way, it is possible to record the test suite as if it were a real web application. Developing the screen mockups and defining the links among them allows also to produce a preliminary but "working" prototype of the web application that can be shown to the stakeholders. This is very useful for detecting, as soon as possible, problems and misunderstandings in the requirements [21]. More in detail, to record the test suite it is necessary to:

1) open with the browser the first HTML mockup of a use case/user story and activate the recording functionality of the selected capture/replay tool;
2) follow the steps described in the use case/user story[4] and replicate them on the HTML mockups (e.g. insert values in the input fields, click links);
3) manually insert the assertions in the generated test scripts.

The order of execution of the test scripts must be defined to:

1) allow the execution of the entire test suite (e.g. "Delete User Test" must be executed after "Add User Test");
2) test each functionality with the corresponding test scripts (e.g. the login functionality must be validated using the "Login Test" and thus such script must be executed before all the other ones requiring a correct user authentication).

D. Web Application Development

Web application development is based on a test-first approach using the previously produced test scripts. The functionalities are implemented/refined following the test suite as a guidance until all tests pass successfully. Finally, stakeholders evaluate the resulting web application and decide whether approving it or moving through a further refinement step. It is important to notice that the web application development can be conducted with any technology – e.g. Ajax – and any development process – e.g. traditional, model-driven (e.g. using WebRatio [1]) or by means of mashups. The only constraint is to use the same text (e.g. for linkText locators) or ID/name attribute values used in the produced mockups. In this way, test scripts recorded on the mockups can be executed also on the real web application without any problem.

[4]notice that in the process displayed in Figure 1 all the artifacts preceding an activity in the flow are available to such activity (e.g. requirements specification and screen mockups are used by the Acceptance Test Suite Development activity).

E. Test Suite Maintainability Improvement

Once the web application has been developed, our approach moves forward to the test suite maintainability step. Test scripts generated through the recording phase often present repeated instructions (e.g. each time the user has to authenticate herself in the system, recorded test scripts include the steps related to the credentials insertion phase) resulting in *code clones*. A good practice is removing code clones by means of refactoring strategies able to encapsulate common test script steps in reusable blocks. After this post-processing step, test scripts are easier to understand and modify and, thus, more maintainable. In this way, a change in a web application functionality will only impact the reusable blocks instead of propagating the change through the entire test suite. As an example, let us consider the following change in the login page: "for security reasons, provide the password twice instead of once"; without a refactoring step, all the test scripts implementing login functionality will need a repair. On the contrary, with a refactoring step, only the reusable blocks will need it.

F. Test Suite Extension

To improve the effectiveness of the test suite and make it more complete, an extension step is needed. Even though test scripts generated as described before can be very useful for developing a web application in ATDD mode, they are not enough to be used for regression purposes because too simple/limited in terms of code coverage and built using hard-coded values (i.e. the ones recorded during the test script development or contained in the screen mockups). To extend a test suite, at least two categories of tools can be adopted to generate input data[5]: *inputs data generator tools* and *fuzzers* [3]. Inputs generator tools are used to generate input data, often stored in files, useful to later populate test scripts. On the contrary, fuzzers are tools able to automatically inject semi-random data into a program (in our case, a web application). Currently, our approach suggests to replace test scripts containing hard-coded values with parametric test scripts (a.k.a. Data-Driven test scripts) able to read previously generated input files containing the input data automatically generated by inputs generator tools. Fuzzers will be considered as future work, as well as smarter input data generators which are able to produce input for test cases covering a larger set of scenarios.

[5]test data generation is the process of creating a set of data for testing the adequacy of new or revised web applications

G. Robustness Improvement

Both during the test suite maintainability improvement and extension phases, new interactions with the web page elements can be added. For instance, new assertions can be included to existing test scripts or additional test scripts can require to interact with web elements not considered in the original acceptance test suite (e.g. new values in tables or lists). In these cases, new locators have to be generated. A good practice is making locators robust as much as possible to reduce test suite maintainability efforts. Indeed, if a locator is fragile (e.g. an absolute XPath or an XPath navigating several levels in the DOM), it will quite surely lead to a test script failure when something changes in the structure of the corresponding web page under test. The *locators fragility problem* can be limited by replacing existing locators with the ones produced by robust locators generator algorithms (e.g. ROBULA+ [14]). Locators robustness improvement can be performed during both the maintainability improvement and extension phases, as shown in Figure 2.

III. The AddressBook Case Study

We decided to evaluate the feasibility of our approach by re-implementing an existing web application. We chose the latest version of AddressBook[6], an addresses/phones/birthdays organizer web application already used in several other studies [23], [6], [24], [13], [12], [10]. To make the evaluation more realistic, we decided to apply the approach depicted in Figure 1 separating the roles: while one of the authors was assigned to the requirements analysis phase, another one was involved in the subsequent phases. For the interested reader, all the produced material (requirements specification, mockups, Selenium IDE ATDD test suite, developed AddressBook prototype, etc.) can be downloaded from http://sepl.dibris.unige.it/2016-ATDD.php

As described in Figure 1, the first step is specifying the requirements for the web application to develop. Thus, for the AddressBook web application, the author responsible for requirements analysis performed an exploratory navigation to reverse engineer its most relevant features, that finally were described by means of a requirements specification composed by 15 use cases. The author playing the role of analyst adopted the method described by Reggio et al. [19].

In such method, use cases are enriched by a glossary of terms to reduce ambiguity and by screen mockups to better explain what an actor can see before/after each step in a scenario. Use cases follow a quite stringent template, and must adhere to a set of constraints for the whole artifacts of the specification (i.e. use cases description, glossary, screen mockups) to improve their consistency. This method was chosen since it helps in better describing scenarios and making performed interactions more explicit and clearer.

Moving forward with the process, the second author selected BlueGriffon[7] tool as the WYSIWYG content editor to design the AddressBook screen mockups, since it presents a simple interface and provides HTML pages to be later used to record test scripts. For each screen mockup to represent, only a subset of the original web elements was reproduced, filtering out those that were not interesting, not useful or not clear (e.g. the web application at hand presents many links that perform similar tasks). Reusable web elements have been identified and shared in every screen mockup (e.g. a common link).

Screen mockups were created by following the order suggested by use cases scenarios, together with some guidelines from the adopted method [19] which helped in the process. In our case, the login page mockup was the first one, followed by the home page mockup and so on, in accordance with the use cases functionalities previously captured. Figure 3 shows a full example, where four generated screen mockups are linked to a use case. The screen mockups were linked to use cases steps any time the system had to show/ask something to the user (e.g. a message or an empty form) or (s)he had to provide some data (e.g. filling a previously shown empty form). The use case given in Figure 3 depicts the screen mockups as hyperlinks to the actual HTML representations produced with BlueGriffon; for example, the *AddEntryPage* hyperlink after step two is a reference to the screen mockup pointed by the arrow. As suggested by the method proposed by Reggio *et al.* [19], a glossary of terms was introduced to reduce their ambiguity; the terms followed by the star symbol in Figure 3 are references to entries in the glossary. For the sake of brevity, just a fragment of the glossary has been shown in the figure.

As suggested by the approach, links among screen mockups were implemented and, when necessary, manually injected with Javascript code to handle input alternatives (e.g. correct/wrong credentials) in order to make them executable. Automatic Javascript injection is part of our future work; currently, we are planning to develop a tool that automatically derives Javascript code from form-based screen mockups and inject them with alternatives checking.

Overall, 35 screen mockups were produced, 18 of them uniquely representing the main sections of the application and the remaining ones derived from the former. Since screen mockups are static, the derived ones were necessary to simulate the behavioural aspect of the web application to develop (i.e. its states). For example, the home page should list all the entries in the system; therefore a new screen mockup to represent it has been created any time data were added, removed or edited.

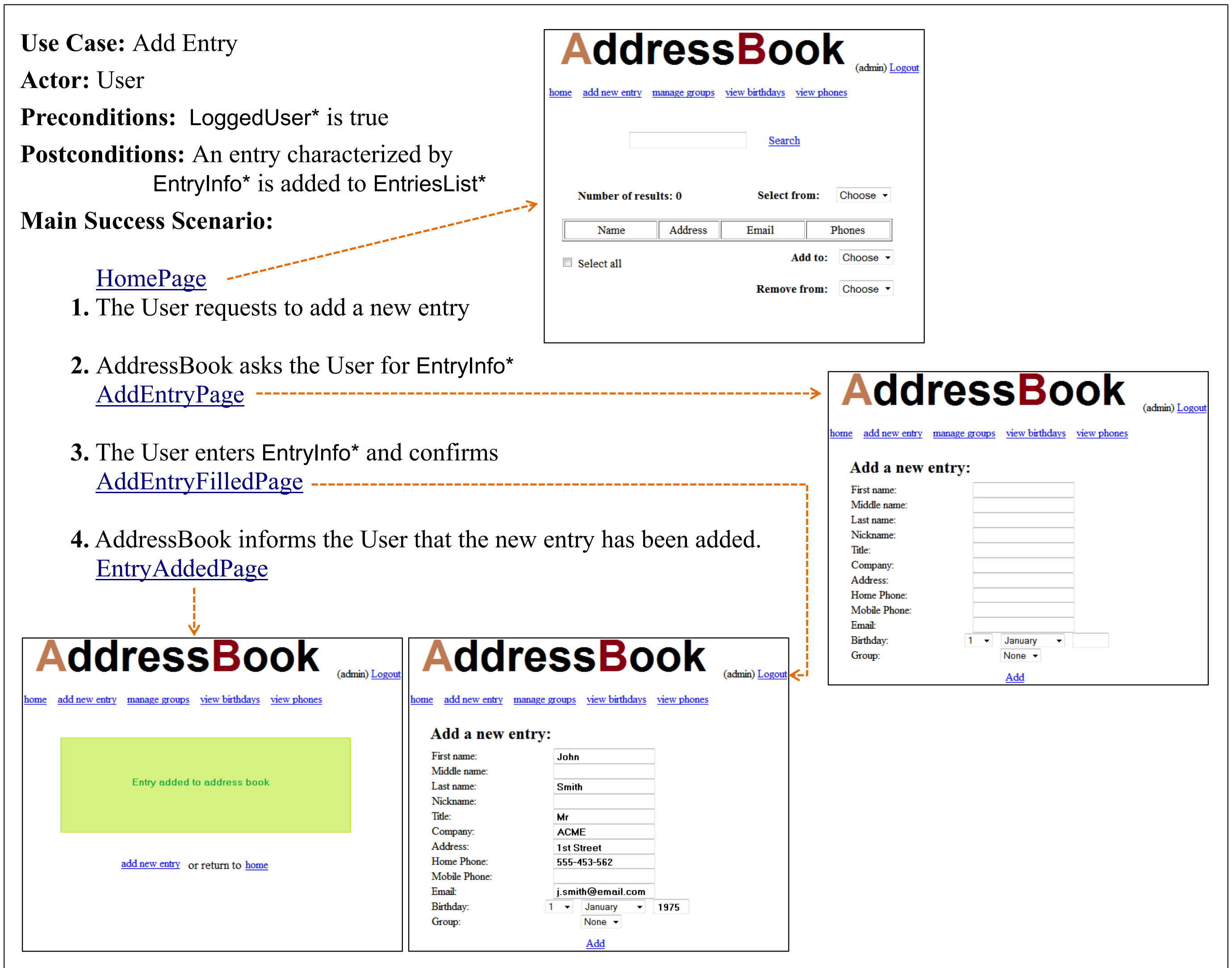
In order to guarantee a correct link between generated test scripts and future web application, the author in charge of development assigned IDs or names to the web elements contained in the HTML screen mockups: 1) on which interactions will occur (e.g. a text field that will be filled) and 2) that will be involved in assertions on data containers (e.g. a label that will show an output message consequent to some user actions).

For the test suite development, the second author selected Selenium IDE, a Firefox plug-in that allows to record, edit, and execute web test scripts. Selenium IDE was chosen among the others capture/replay tools since it is largely used [11], it has a simple interface, and offers a large variety of useful extensions. Adopting a capture-replay DOM-based tool like Selenium IDE

[6] https://sourceforge.net/projects/php-addressbook/

[7] http://www.bluegriffon.org/

Fig. 3. The use case for adding a new entry, adorned with screen mockups and a fragment of the glossary



Use Case: Add Entry
Actor: User
Preconditions: LoggedUser* is true
Postconditions: An entry characterized by EntryInfo* is added to EntriesList*
Main Success Scenario:

HomePage
1. The User requests to add a new entry

2. AddressBook asks the User for EntryInfo*
AddEntryPage

3. The User enters EntryInfo* and confirms
AddEntryFilledPage

4. AddressBook informs the User that the new entry has been added.
EntryAddedPage

First name:	John
Middle name:	
Last name:	Smith
Nickname:	
Title:	Mr
Company:	ACME
Address:	1st Street
Home Phone:	555-453-562
Mobile Phone:	
Email:	j.smith@email.com
Birthday:	1 January 1975
Group:	None

Glossary (fragment)

Data Entry:
- EntryInfo: all the characterizing info for an entry: Firstname, Middlename, Lastname, Nickname, Title, Company, Address, Phones*, Email*, Birthday*, AssignedGroup*

....

System State:
- LoggedUser: true if the User is logged into AddressBook, false otherwise
- EntriesList: the list of entries into AddressBook, each one characterized by EntryInfo*

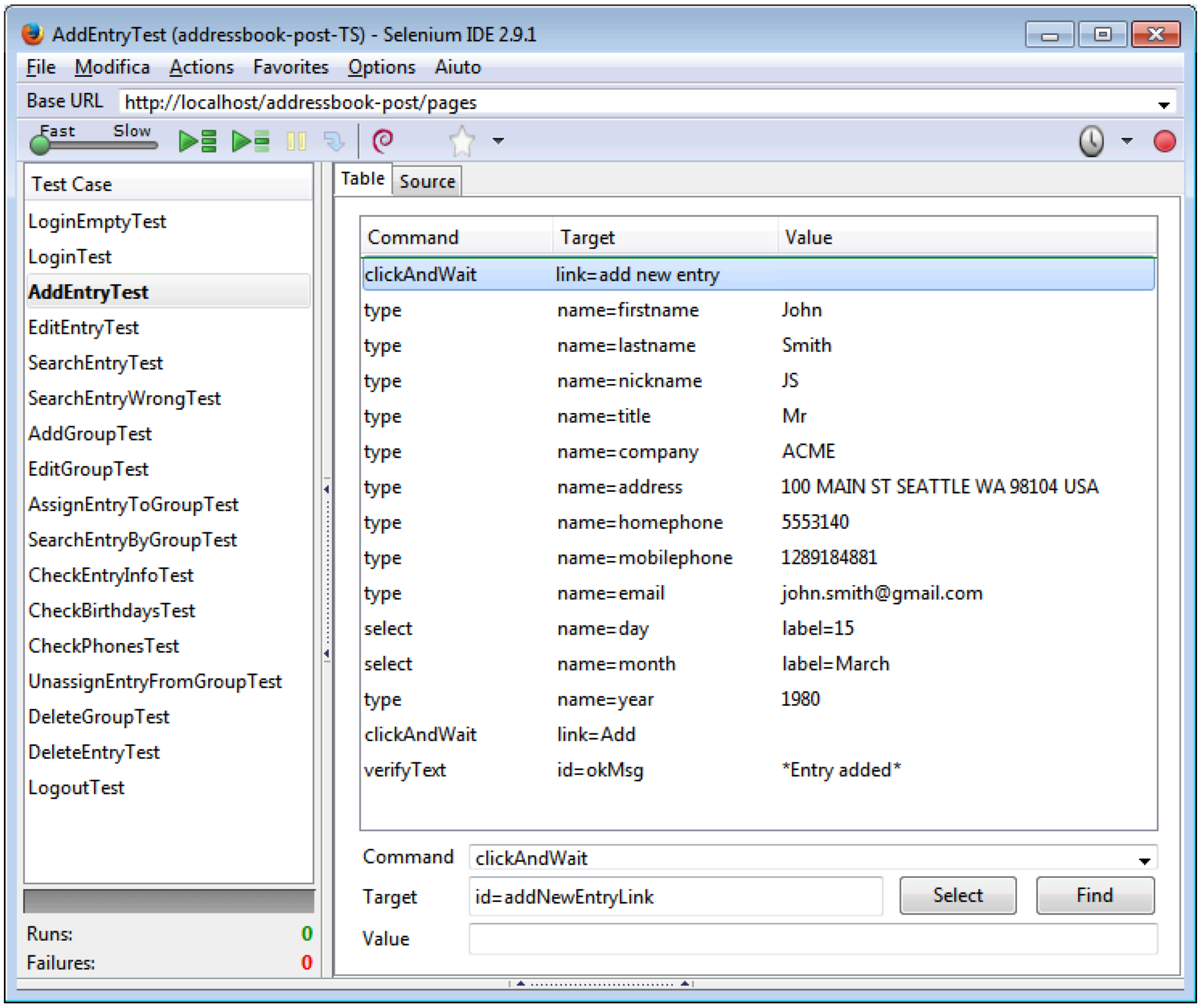
....

allows to pay little attention to the mockups graphical aspect and focus on the user interaction, with clear advantages in terms of effort required for creating the test scripts [9]; for this reason we avoided both visual or programmable [11] web testing tools, such as Sikuli[8] or Selenium WebDriver[9] respectively.

The adoption of both use cases and screen mockups allowed the second author to record test scripts during the process, since interactions naturally followed the use cases scenarios. Moreover, screen mockups enriched the comprehension of which interactions and data were needed. For example, *AddEntryPage* and *AddEntryFilledPage* hyperlinks from Figure 3 refer to screen mockups that represent a form-based web page, before

[8]http://www.sikuli.org/
[9]http://www.seleniumhq.org/projects/webdriver/

Fig. 4. The test script for adding a new entry


AddEntryTest (addressbook-post-TS) - Selenium IDE 2.9.1
File Modifica Actions Favorites Options Aiuto
Base URL http://localhost/addressbook-post/pages
Fast Slow
Test Case
Table Source
Select Find
Runs: 0
Failures: 0

LoginEmptyTest
LoginTest
AddEntryTest
EditEntryTest
SearchEntryTest
SearchEntryWrongTest
AddGroupTest
EditGroupTest
AssignEntryToGroupTest
SearchEntryByGroupTest
CheckEntryInfoTest
CheckBirthdaysTest
CheckPhonesTest
UnassignEntryFromGroupTest
DeleteGroupTest
DeleteEntryTest
LogoutTest

Command	Target	Value
clickAndWait	link=add new entry	
type	name=firstname	John
type	name=lastname	Smith
type	name=nickname	JS
type	name=title	Mr
type	name=company	ACME
type	name=address	100 MAIN ST SEATTLE WA 98104 USA
type	name=homephone	5553140
type	name=mobilephone	1289184881
type	name=email	john.smith@gmail.com
select	name=day	label=15
select	name=month	label=March
type	name=year	1980
clickAndWait	link=Add	
verifyText	id=okMsg	*Entry added*

Command: clickAndWait
Target: id=addNewEntryLink
Value:

and after filling it with data, so they suggest which input the tester may use to fill the textfields.

Each test script was generated starting from use cases scenarios. Initially, the second author had to open the first screen mockup of each use case, according to the listed pre conditions. For instance, if a pre condition states that a list of entries is shown, then the test script should start from the screen mockup where that list is actually displayed. Then, the recorded interactions on the web elements followed the scenarios steps, while assertions were manually added, guided by the post conditions described in the use cases. As an example, if a post condition states that an entry is added to a list, then that entry should be visible in the list and the system should inform about the completion of the operation (to be checked by means of an assertion).

The final acceptance test suite was made up of 17 simple test scripts. A sample one, expressed in the Selenese[10] language, is shown in Figure 4. Here the test script performs some interactions (see the column Command in the Selenium IDE interface shown in Figure 4) on the web elements, which are identified by their locators (Target column), providing some input data (Value column). In the example, the test script clicks on the link identified by the "add new entry" textual locator (more in detail the text of a link) to access the page where the entry can be added. Then, it performs some interactions on form fields (i.e. textfields and drop down menu), to enter/select input data for the new entry; the web elements are located by the name property. Finally, it confirms the insertion by clicking on the "Add" link. The assertion is manually added and checks whether the confirmation message is shown in a label located by a specific id.

The previously generated screen mockups representing unique aspects of AddressBook (18 out of 35) were adopted to re-develop the web application (17 mockups were excluded from the process because they were just mere instantiations, i.e. replications with different data). According to the ATDD paradigm, development was guided by test scripts, step by step.

[10]each Selenese line is a triple: (command, target, value)

Fig. 5. The Javascript rollup rule for adding a new entry

```
var manager = new RollupManager();

manager.addRollupRule({
  name: 'add_entry',
  description: 'add a new entry',
  args: [],
  commandMatchers: [],

  getExpandedCommands: function(args) {
    var commands = [];
    commands.push({
      command:'clickAndWait', target:'link=add new entry'
    });
    commands.push({
      command:'type', target:'name=firstname', value:'${firstname}'
    });
    commands.push({
      command:'type', target:'name=lastname', value:'${lastname}'
    });
    commands.push({
      command:'type', target:'name=nickname', value:'${nickname}'
    });
    commands.push({
      command:'type', target:'name=title', value:'${title}'
    });
    commands.push({
      command:'type', target:'name=company', value:'${company}'
    });
    commands.push({
      command:'type', target:'name=address', value:'${address}'
    });
    commands.push({
      command:'type', target:'name=homephone', value:'${homephone}'
    });
    commands.push({
      command:'type', target:'name=mobilephone', value:'${mobilephone}'
    });
    commands.push({
      command:'type', target:'name=email', value:'${email}'
    });
    commands.push({
      command:'select', target:'name=day', value:'${day}'
    });
    commands.push({
      command:'select', target:'name=month', value:'${month}'
    });
    commands.push({
      command:'type', target:'name=year', value:'${year}'
    });
    commands.push({
      command:'clickAndWait', target:'link=Add'
    });
    commands.push({
      command:'verifyText', target:'id=okMsg', value:'*Entry added*'
    });
    return commands;
  }
});
```

As expected, they failed at the first run so, in order to pass the tests and fulfil the expected features, the author in charge for the development had to implement AddressBook dynamic behaviours (mostly through PHP code to handle with actual data and navigation). The test scripts failures were easy to detect and fix since Selenium IDE provides quite explicative messages. Following the process, he had to switch back to mockups and test suite development just a couple of times, due to minor changes in the GUI or in interactions.

At the end, the recorded test suite led successfully to a preliminary but working AddressBook web application. Notice that no changes to the web elements were needed, since the same IDs/names of the screen mockups were preserved. From this point, layout could be improved by adding CSS to enrich the GUI of the web application, with no impact on test scripts.

Fig. 6. The structure of the XML data file for adding a new entry

```
<testdata>

    <vars  firstname   = "..."
           lastname    = "..."
           nickname    = "..."
           title       = "..."
           company     = "..."
           address     = "..."
           homephone   = "..."
           mobilephone = "..."
           email       = "..."
           day         = "..."
           month       = "..."
           year        = "..."
    />

</testdata>
```

The second author then proceeded to enhance the test suite, as shown in Figure 2. To factorize test scripts, he used the Selenium IDE *rollup* command to group repeated sequences (i.e. clones) of Selenese instructions and reuse them across different test scripts in the test suite. The rollup command refers to a Javascript file that stores the shared Selenese triples (i.e. command, target, value) as a set of rules to be called any time they must be executed. In Figure 5 a Javascript rollup rule is shown. It includes all the interactions with the web application needed to add a new entry (i.e. clicking the link to visit the page where the form is located, inserting the data into the form, and confirming it, as shown also in test script of Figure 4). Thus, this rule can be referred inside Selenium test scripts through the specified property name. In this way, interactions upon highly used web elements, such as text fields inside a login form, are contained in the file, therefore any change that may influence those elements will impact just the rule, and eventually test scripts become more robust and readable (e.g. as shown in Figure 7, the single *rollup add_entry* instruction replaces those listed in Figure 4).

We believe that the adoption of rollup rules can provide substantial advantages for what concerns the effort of maintaining the test suites. For instance, Leotta *et al.* [11] show the benefits of adopting the Page Object pattern[11], a quite popular web test design pattern, which aims at improving the test case maintainability and at reducing the duplication of code. A page object is a class that represents the web page elements and that encapsulates the features of the web page into methods. With the page object pattern, each method can be reused more times in a test suite. Thus, a change at the level of the page object can repair more than one test case at once. We believe that, from the maintainability point of view, having rollup rules or page object methods is quite similar.

[11]http://martinfowler.com/bliki/PageObject.html

In total, 14 rollup rules have been defined, saving 51 LOCs due to repeated instructions. Clearly, the benefits from using rollup rules depend mostly on the test suite size and on the number of times a block of instructions is reused among the test scripts (e.g. adding a new user). This becomes particularly evident in the next phase (Test Suite Extension), where the test scripts are re-executed several times with different data.

Since generated acceptance test scripts contained hard-coded input data by construction (see Figure 4), the second author transformed them in parametric test scripts, relying on XML datasets created by the GenerateData[12] input generator, and executed them with Sel Blocks[13], a Selenium IDE extension able to execute parametric test scripts. Even though the generated data were unable to cover all possible scenarios which may originate from feeding a form with input data, it was enough to drastically augment the test suite produced in the previous phase of the approach. We chose to keep also the old hard-code based test scripts since they were still useful for testing purposes (the meaningful data recorded are able to cover, at least once, each existing scenario). To make multiple data understandable by Sel Block, just few changes in their XML structure were needed; in Figure 6 the accepted test data template is shown, where the *vars* tag represents a random instance with its attributes. Furthermore, test scripts were enriched by additional instructions, such as loops that cycle across the XML entries or parameterized commands that take in consideration the multiplicity of the given data (e.g. clicking on the right edit link associated to the currently selected user). Consequently, parameterized assertions were manually introduced as well. At the end, the test suite which was originally made up of 17 hard-coded test scripts have been augmented to 487 executable test scripts, since some of the original ones were parameterized with several different input/expected values previously stored in XML datasets. In Figure 7, the enhanced version of the test script shown in Figure 4 is given. The test script is now evidently shorter, since all the instructions to add a new entry in the system are enclosed in the rollup command (Figure 5), while the *forXml* and *endForXml* commands are used to iterate across the provided XML file.

Finally, the second author applied the ROBULA+ algorithm [14] to generate robust locators for the new web elements used in the additional test scripts and for the further assertions added to the existing ones, in all cases in which Selenium IDE relied on fragile navigational XPath locators. In this way, it was possible to improve test scripts robustness by means of more robust XPath locators (e.g. this is particularly useful in case of dynamic DOM structures, such as tables, containing changing data that cannot be easily provided of meaningful attributes). The locators generated by ROBULA+ are, in general, by far more robust than the ones produced by Selenium IDE (63% fragility reduction) [14].

[12]http://www.generatedata.com

[13]https://addons.mozilla.org/it/firefox/addon/selenium-ide-sel-blocks/

A. Limitations and Future Improvements

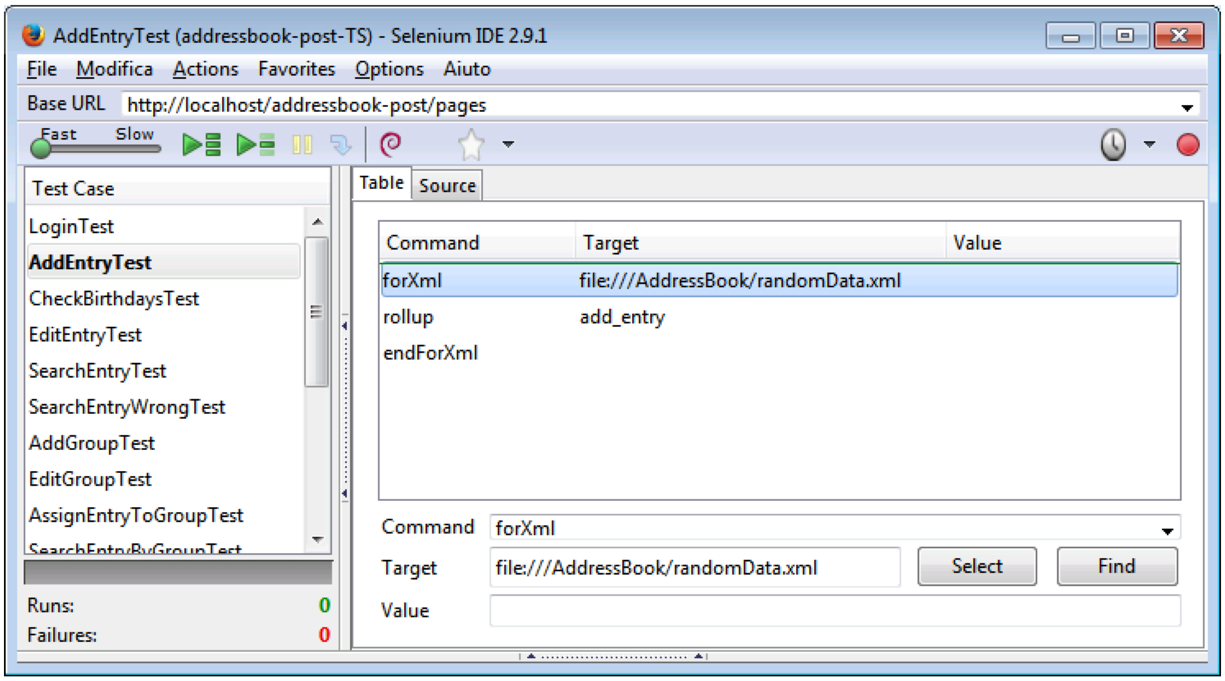
First, we found the approach feasible and simple to apply for small and medium sized web applications, but probably the same process would be quite costly for big web applications. In such cases, we believe, it may lead to a high number of screen mockups, and so to a relevant effort from the mockups designers point of view. Moreover, to limit the number of screen mockups to be produced, interactions upon web elements must be reduced as well. Unfortunately, this has the consequence to limit the number of functionalities tested and the coverage of the test scripts produced. Second, to benefit from the approach and keep test scripts runnable, mockups designers must preventively associate robust hooks, such as for instance meaningful IDs or names, to web elements (even to labels that may potentially express some useful data to assert). This can be a cumbersome task. Third, reproducing the navigation among mockups and injecting Javascript code to simulate dynamic behaviours can also be a tedious and time-consuming task, if manually performed. Moreover, test suite enhancement requires additional Javascript code to introduce rollup rules, while the generated random data needs some improvements to affect more scenarios and provide smarter datasets. Techniques to automatically generate assertions are also an important topic to investigate. For this reasons, in the future we intend to provide a tool that combines the phases of development (Figure 1) with the ones of test suite improvement and extension (Figure 2) and automates all the steps of the proposed approach.

B. Lesson Learnt

The adoption of the proposed approach to the development of the AddressBook application resulted in some lesson learnt:

- the main lesson we learnt is that the proposed approach is applicable to the development of medium-size web applications with a limited effort. We indeed believe that the process is generalizable also for other web applications of similar complexity.
- we have experimented how requirements expressed by means of use cases and screen mockups [19] are helpful to guide test suite development, since their scenarios naturally describe user interactions more than concise user stories may do. Extensions in use cases clearly suggest what must be tested and which scenarios should be considered. Pre and post conditions allow to determine, respectively: (1) the starting point from which to record a test script, and (2) the conditions/properties that must be checked by a test script.
- the practice of adding meaningful hooks to web elements turns out to be useful, since it guarantees to have robust locators during software evolution and maintenance, even more when an algorithm like ROBULA+ is subsequently applied. This algorithm showed its utility in formulating complex assertions based on XPath locators involving nested elements (e.g. as in the case of an entry in a table).
- the adoption of an input generator tool (and of the Selenium IDE extensions) can make test suite enhance-

Fig. 7. The enhanced test script for adding a new entry



Command	Target	Value
forXml	file:///AddressBook/randomData.xml	
rollup	add_entry	
endForXml		

Command forXml
Target file:///AddressBook/randomData.xml
Value

ment easier and more effective, since tester can simply record interactions from mockups and then improve their complexity with loops, covering a larger variety of interesting input combinations through multiple runs on randomly generated data.

- it is not a problem to pay a little more effort in screen mockups creation, given that they can be then reused as the basis for the development of the web application GUI.

IV. Related Works

Hellmann *et al.* [8] present a Test Driven Development approach for GUI-based applications, where low fidelity prototypes are sketched and linked together through event handlers to make the navigation possible. In a second time, interactions on prototypes are recorded to produce test scripts able to drive the development. While the idea is similar to the one proposed in our paper, some differences exist: context, adopted technologies, requirements analysis phase and emphasis on test scripts robustness.

Olek *et al.* [18] propose a coding language (Test Description Language) for defining test scripts following use cases steps, by recording interactions on low detailed sketches. Similar to us, use cases are used as a starting point for test scripts definition. While our approach is tool independent and can be instantiated by practitioners in several ways, in the approach proposed by Olek *et al.* the aid of specific tools is essential and manual intervention is required to code complex interactions and to translate test descriptions into executable scripts.

In the context of Model-Driven Web Engineering (MDWE), Rivero *et al.* [22] propose an iterative agile-MDWE process based on mockups to support web applications development. HTML mockups are generated from user stories, tagged to explicit widgets semantic and turned into MDWE models to generate code. The main difference with respect to our work is that the work of Rivero *et al.* is not based on the ATDD paradigm. In their paper, user stories are the basis to design screen mockups, although they do not support the testing phase.

Mugridge [16] developed an extension to the Fit framework[14] to improve expressiveness of story tests (i.e. fixtures workflows based on user stories), automatically coding them into executable test scripts. Similarly to us, user stories can guide test scripts definition and execution. However, the paper does not focus on web domain and testing evolution. Test scripts need fixtures tables and an actual system to run, while in our approach they can be recorded and executed directly on screen mockups.

Besson *et al.* [2] propose an ATDD approach for web applications development based on user stories. Functionalities are mapped into a graph, where each path represents a testing scenario as a navigation through pages. Testing scenarios are then validated by the customer and subsequently (semi-automatically) transformed into executable test scripts. Conversely to Besson *et al.*, our approach does not require the graph structure, which is substituted by a simpler recording phase of interactions on screen mockups. Therefore, test scripts are easier to get and to maintain during web application evolution.

[14]http://fit.c2.com/

V. Conclusion and Future Work

In this work, we have proposed a novel approach for developing web applications following the ATDD paradigm. The novelty concerns the usage of screen mockups and capture/replay testing tools for easily generating acceptance test scripts guided by requirements specifications and able to drive the development of the target web application.

The approach has been successfully applied on a sample application to show its feasibility. The main lesson learnt is that the proposed approach is applicable to the development of medium-size web applications with a limited effort. The obtained encouraging results on the selected sample application suggest that more comprehensive studies (comparative experiments, case studies, and evaluation of the actual industrial applicability) should be conducted to gather feedbacks on the effectiveness and usefulness of our approach.

As future work, we want to continue this line of research, overcoming the approach limitations and building a full-fledged tool able of implementing the steps composing the proposed approach.

References

[1] R. Acerbis, A. Bongio, M. Brambilla, and S. Butti. Model-driven development based on OMG's IFML with WebRatio web and mobile platform. In P. Cimiano, F. Frasincar, G.-J. Houben, and D. Schwabe, editors, *Proceedings of 15th International Conference on Web Engineering (ICWE 2015)*, volume 9114 of *LNCS*, pages 605–608. Springer, 2015.

[2] F. M. Besson, D. M. Beder, and M. L. Chaim. An automated approach for acceptance web test case modeling and executing. In *Proceedings of 11th International Conference on Agile Software Development (XP 2010)*, volume 48 of *LNBIP*, pages 160–165. Springer, 2010.

[3] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[4] D. Clerissi, M. Leotta, G. Reggio, and F. Ricca. A lightweight semi-automated acceptance test-driven development approach for web applications. In A. Bozzon, P. Cudré-Mauroux, and C. Pautasso, editors, *Proceedings of 16th International Conference on Web Engineering (ICWE 2016)*, volume 9671 of *Lecture Notes in Computer Science*, pages 593–597. Springer, 2016.

[5] G. Downs. Lean-agile acceptance test-driven development: Better software through collaboration by Ken Pugh. *ACM SIGSOFT Software Engineering Notes*, 36(4):34–34, 2011.

[6] M. Hammoudi, G. Rothermel, and P. Tonella. Why do record/replay tests of web applications break? In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2016)*, page (in press). IEEE, 2016.

[7] H. R. Hartson and E. C. Smith. Rapid prototyping in human-computer interface development. *Interacting with Computers*, 3(1):51–91, 1991.

[8] T. D. Hellmann, A. Hosseini-Khayat, and F. Maurer. Test-driven development of graphical user interfaces: A pilot evaluation. In *Proceedings of 12th International Conference on Agile Software Development (XP 2011)*, pages 223–237, 2011.

[9] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proceedings of 20th Working Conference on Reverse Engineering (WCRE 2013)*, pages 272–281. IEEE, 2013.

[10] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Visual vs. DOM-based web locators: An empirical study. In M. W. Sven Casteleyn, Gustavo Rossi, editor, *Proceedings of 14th International Conference on Web Engineering (ICWE 2014)*, volume 8541 of *Lecture Notes in Computer Science*, pages 322–340. Springer, 2014.

[11] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Approaches and tools for automated end-to-end web testing. *Advances in Computers*, 101:193–237, 2016.

[12] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Automated generation of visual web tests from DOM-based web tests. In *Proceedings of 30th ACM/SIGAPP Symposium on Applied Computing (SAC 2015)*, pages 775–782. ACM, 2015.

[13] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Using multi-locators to increase the robustness of web test cases. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, pages 1–10. IEEE, 2015.

[14] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, 2016.

[15] A. McDonald and R. Welland. Agile web engineering (AWE) process. Technical report, University of Glasgow, 2001.

[16] R. Mugridge. Managing agile project requirements with storytest-driven development. *IEEE Software*, 25(1):68–75, 2008.

[17] M. O'Docherty. *Object-Oriented Analysis and Design: Understanding System Development with UML 2.0*. Wiley, 1 edition, June 2005.

[18] L. Olek, B. Alchimowicz, and J. R. Nawrocki. Acceptance testing of web applications with test description language. *Computer Science (AGH)*, 15(4):459, 2014.

[19] G. Reggio, M. Leotta, and F. Ricca. A method for requirements capture and specification based on disciplined use cases and screen mockups. In *Proceedings of 16th International Conference on Product-Focused Software Process Improvement (PROFES 2015)*, volume 9459 of *Lecture Notes in Computer Science*, pages 105–113. Springer, 2015.

[20] G. Reggio, F. Ricca, and M. Leotta. Improving the quality and the comprehension of requirements: Disciplined use cases and mockups. In *Proceedings of 40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2014)*, pages 262–266. IEEE, 2014.

[21] F. Ricca, G. Scanniello, M. Torchiano, G. Reggio, and E. Astesiano. Assessing the effect of screen mockups on the comprehension of functional requirements. *ACM Transactions on Software Engineering and Methodology*, 24(1):1–38, 2014.

[22] J. M. Rivero, J. Grigera, G. Rossi, E. R. Luna, F. M. Simarro, and M. Gaedke. Mockup-driven development: Providing agile support for model-driven web engineering. *Information & Software Technology*, 56(6):670–687, 2014.

[23] K. P. Shabnam Mirshokraie, Ali Mesbah. Atrina: Inferring unit oracles from GUI test cases. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2016)*, page (in press). IEEE, 2016.

[24] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. Clustering-aided page object generation for web testing. In A. Bozzon, P. Cudré-Mauroux, and C. Pautasso, editors, *Proceedings of 16th International Conference on Web Engineering (ICWE 2016)*, volume 9671 of *Lecture Notes in Computer Science*, pages 132–151. Springer, 2016.