Computer Science • 15 (4) 2014 http://dx.doi.org/10.7494/csci.2014.15.4.459

Łukasz Olek
Bartosz Alchimowicz
Jerzy Nawrocki

ACCEPTANCE TESTING OF WEB APPLICATIONS WITH TEST DESCRIPTION LANGUAGE

Abstract *Acceptance tests are usually created by a client after a part of a system is implemented. However, some methodologies propose the elaboration of test cases before implementing a system. This approach increases the probability of system implementation that fulfills requirements, but may be problematic for customers and testers. To allow acceptance testing in such conditions, we propose to define test cases by recording them on an interactive mockup (a low detailed user-interface prototype). The paper focuses on Test Description Language, a notation used to store test cases.*

Keywords acceptance test case, use case, mockup, Test Description Language

1. Introduction

The aim of acceptance testing is to make sure that a target system meets acceptance criteria [8]. Traditionally, tests are created by a client after a part of a system is implemented. This is natural, because test cases can be executed only when the corresponding features are ready. However, agile methodologies (see e.g., [3]) propose a different approach: test cases should be elaborated before implementation. Unfortunately, this may be problematic for customers and testers. Without seeing an application, clients may create tests that are inadequate and contain mistakes. Testers, due to a lack of details, may not be able to implement these test cases. Additionally, if the requirements are modified, one needs to check all test cases and update them – and this is a task which generates additional costs. An approach that provides a link between tests and other artifacts in a project while being easy to use and maintain by IT-laymen would be beneficial here.

To test a web-application, one may use a number of tools. One of them is Selenium [4, 9]. It provides a record/playback tool which allows testing without learning the domain-specific language used to write the tests. Unfortunately, recording a test necessitates a working web application. It is possible to write tests without an application (in the form of an HTML file or unit tests), but this requires additional technical knowledge.

Another approach is using FIT tables [10, 1, 7]. FIT allows its users to specify test cases as tables and then provide a code (called a fixture) to execute these tables on a target system. The actual test results are directly visible in the table as particular cells marked with green (success) or red (failure). However, this approach causes many problems and is not easy to employ in practice. Firstly, it is difficult to comprehend (not to mention to write) FIT tables for people without an IT education. Tables do not represent the system visually and require a considerable effort in abstract thinking to be well understood [11, 17]. Melnik observed: *The test pages were so long (dozens of screen scrolls) and wide (dozen of columns), that navigating and deriving meanings from the test was extremely difficult* [11]. Secondly, a substantial effort is required to specify test cases as tables. Melnik's participants *were surprised at the amount of work that is involved in specifying these tests.* Thirdly, technology experts (programmers or testers) feel an initial resistance to this method, due to the effort needed to invest in programming the fixtures.

Cucumber is another tool that may be used for acceptance testing [20]. One can describe an interaction between a user and a web application in plain text using a domain-specific language called Gherkin. To check whether the described actions can be accomplished using an application, testers need to write code that will interpret a given text and run commands on a web page (enter a text, click an element, etc). This tool also supports tables like FIT (with all their advantages and disadvantages). A working application is not required, but without highly-developed abstract thinking, it may be hard to create tests (similar to FIT).

The approaches presented have one common disadvantage. Since there is no connection between requirements and test cases, maintainability is a big issue. When a requirement changes, one needs to browse all tests and update them manually. One could try to use a traceability matrix to describe relations between tests and requirements; however, it is well known that programmers do not like to use documentation [16]. Thus, it may become outdated.

In order to elaborate test cases before implementing a web application in an easy-to-create and -maintain way, we propose a simple approach to defining test cases by recording them on an interactive mockup. An interactive mockup is a kind of user-interface prototype [6, 18, 14, 19] that can observe user actions and record them as a test case. It is a low-fidelity prototype [15] that presents only the structure of screens without any visual details, such as exact fonts, colors, or precise component layout. The mockup presents user interactive screens. The user interacts with them in a similar fashion as a real system. User actions are being recorded as a test case and stored in Test Description Language. Each screen is presented to the user in two modes: user-action mode and verification-point mode. In user-action mode, a screen works very much like a real system, and the user can fully interact with its components. In verification-point mode, a screen allows the user to define expected values of all components. In this mode, the user can also set expected values of semi-compound and compound components (e.g., add table rows). The order in which mockups are shown is generated on the basis of steps (in use cases) linked with screen designs. While recording a test case, one can set which tests need to be executed first – this allows for putting a target system into its required state.

Test Description Language is associated with use cases (defined using FUSE [12]) and screen designs (in ScreenSpec [13]). Such associations allow us to automatically check if test cases are in sync with requirements (use cases and screens) as well as automatically adjust test cases when requirements change.

The focus of this paper is on Test Description Language with respect to traditional request-response web applications, without the interactive elements of Web 2.0.

This paper is organized as follows: In Section 2, an example of an acceptance test case is presented. Section 3 describes Test Description Language. Next, in Section 4, preconditions in test cases are examined. An early evaluation is detailed in Sections 5. Finally, in Section 6, the most important findings are discussed.

2. A simple test-case example

An example described in this section illustrates the process of recording a test case on a mockup. The mockup is built from use cases (in FUSE) and screen designs (in ScreenSpec). A user records test cases, which are stored in Test Description Language in the Test Repository.

As the example, we will consider a simple application that can send e-mails to a chosen recipient list: students or staff. Figure 1 presents a use case that describes this

```
UC1. Post a message to a mailing list
Main Scenario:
1. Author writes a message [MessageEditor].
2. System previews the message [MessagePreview]
3. Author selects a mailing list [MessagePreview].
4. System sends the message to all recipients and
   displays a summary [SendingInfo].
Extensions:
3.A. ALT: Author wants to correct the message.
3.A.1. Author goes back to the message editor
       [MessagePreview].
3.A.2. System returns to Step 1.
```

Figure 1. A use case that describes a feature of posting a message to a mailing list.

functionality: a user prepares a message to be sent, then he/she chooses a recipient list. Finally, the system sends messages and presents a summary.

Use case UC1 (Fig. 1) is associated with three screens: `MessageEditor`, `MessagePreview`, and `SendingInfo`. Their definition in ScreenSpec and visual representation are presented in Figure 2.

Let us consider the simplest test scenario for this use case: sending a message to one of the recipient lists (the sole main scenario). Let us assume that a user wants to send a message entitled "My first email" with content "The content of my first email" to the 'Students' recipient list. This list contains 9 valid email addresses and 1 invalid. The corresponding test case is presented in Table 1. Steps of the test case describe both verification points and user actions.

A reader can observe that, for each step of the exemplary test case (Table 1), there are verification points or user actions specified. For some steps, there are both. Which rule determines if a step will have a verification point or user actions specified? Roughly speaking, for each screen that is displayed by a system to a user, we need to have an ability to specify a verification point and user actions. When the same screen is shown in consecutive steps, a verification point can be recorded in one step and user actions in another (UC1.2 and UC1.3 in the example from Table 1).

A user records a test case using an interactive mockup. The mockup presents interactive screens in an order determined by a test scenario. In the exemplary test case (Table 1), three screens will be presented to a user, each in two modes: verification point and user-action modes. In step 2, for instance, a user will see the `MessagePreview` in verification-point mode (see Fig. 3a). In this mode, values of all `DYNAMIC_TEXT` components (`Subject`, `Content`) can be set. For instance, a `DYNAMIC_TEXT` component that represents a read-only text on a screen is shown in verification-point mode as an edit box (Fig. 3a), which allows a value to be entered. This value will be used as the expected value for this particular component in a test case. The verification point described in step 2 (Table 1) can be recorded by setting expected values of screen components (Fig 3b.).

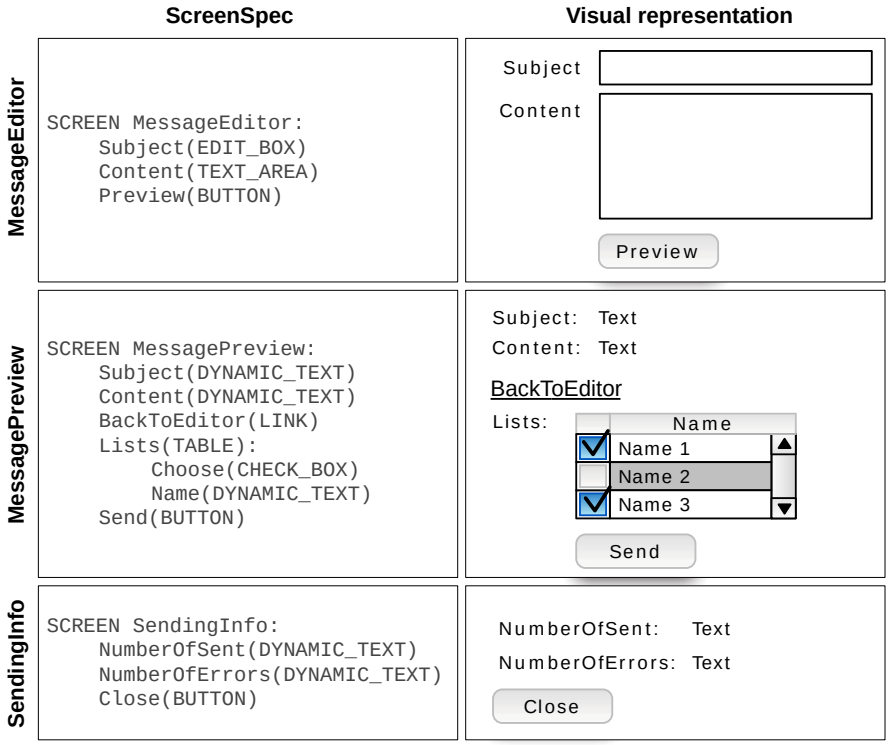


Figure 2. ScreenSpec definitions of screens used in the use case in Figure 1 with a their visual representation.

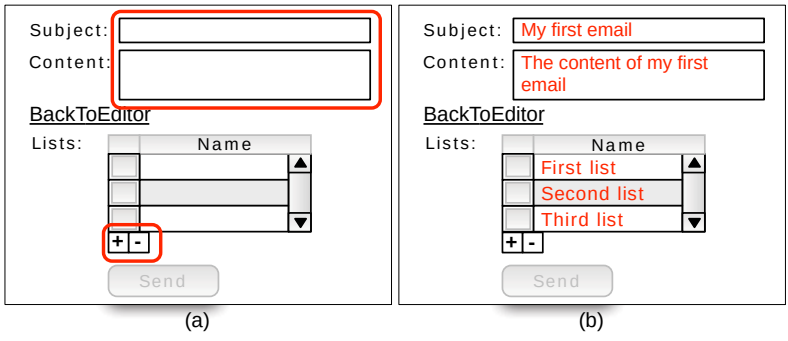


Figure 3. `MessagePreview` screen (see Fig. 2) in verification point mode: (a) before setting expected values, (b) after setting expected values.

Table 1

The exemplary test case.

Step	Verification Point	User Actions
`UC1.1. Author writes a message [MessageEditor].`	**Subject** is empty. **Content** is empty.	Enter "My first email" in **Subject** Enter "The content of my first email" in **Content**. Click **Preview**.
`UC1.1. System previews the message [MessagePreview]`	**Subject** is "My first email". **Content** is "The content of my first email". **First list** checkbox is unchecked. **Second list** checkbox is unchecked. **Third list** checkbox is unchecked.	
`UC1.3. Author selects a mailing list [MessagePreview]`		Check **First list**. Click **Send**.
`UC1.4. System sends the message to all recipients and displays a summary [SendingInfo]`	**NumberOfSent** equals 9. **NumberOfErrors** equals 1.	Click **Close**.

Figure 4. `MessagePreview` screen in user action mode. All components are interactive: a user can choose checkboxes, click buttons, etc.

After setting the expected values of a verification point, the screen is switched into user-action mode (Fig. 4). In this mode, screens are interactive and somehow resemble the target system. For the purpose of the test case, a user chooses the "Students" mailing list and clicks "Send". Then, the user can move to the next step of the test case.

Test cases are recorded with interactive mockups but stored in a special language called Test Description Language (TDL). Figure 5 presents the exemplary test case coded in TDL. Each test case starts with a header (`TESTCASE TC1`) followed by a declaration of a precondition (`PRECONDITION`), which points to a precondition-states dictionary (see Section 4). A test case is a sequence of test steps (`TESTSTEP`) assigned

```
TESTCASE TC1. Sending a~simple message to mailing lists.
        PRECONDITION Pre01
        TESTSTEP UC1.1(MessageEditor):
                CHECK Subject=""
                CHECK Content=""
                SET Subject="My first email"
                SET Content="Mistaken content of my first email"
                CLICK Preview
        TESTSTEP UC1.2(MessagePreview):
                CHECK Subject="My first email"
                CHECK Content="The content of my first email"
                CHECK "First list"=NOT CHECKED
                CHECK "Second list"=NOT CHECKED
                CHECK "Third list"=NOT CHECKED
        TESTSTEP UC1.3(MessagePreview):
                CHECK "First list"=CHECKED
                CLICK Send
        TESTSTEP UC1.4(SendingInfo):
                CHECK NumberOfSent=9
                CHECK NumberOfErrors=1
                CLICK Close
```

Figure 5. The exemplary test case written using Test Description Language.

to use-case steps and screens (`TESTSTEP UC1.2 (MessageEditor)`). Test steps can describe both the verification point and user actions, but not necessary both for each step. Verification points start with the `CHECK` keyword followed by the component's name and expected value. For instance, assertion `CHECK Subject=""` makes sure that component `Subject` will have an empty value. User actions start with action keywords: `SET`, `CLICK`. Action `SET Subject="My first email"` means that value `My first email` will be entered into component `Subject`.

This simple example does not present more-complex language structures, such as semi-compound components nor compound components. These elements are described in details in Section 3.

3. Test Description Language

The basic structure of a test case is presented in Figure 6. The first line declares the new test case. The next line declares a precondition that should be satisfied before executing the test case (more details about preconditions can be found in Section 4). The following lines describe the steps of a test case. Each test case is a sequence of test steps (`TESTSTEP`), linked with use-case steps (by their tags) and screens (by their names). Since use-case steps are already linked with screens (in FUSE), it may look a bit redundant. However, the screens associated with particular steps may change, so this information is important for checking the consistency of tests and requirements.

The grammar of TDL is available online[1].

[1] `http://www.se.cs.put.poznan.pl/projects`

```
TESTCASE TC1. Sending a simple message to mailing lists.
     PRECONDITION PRE01
     TESTSTEP UC1.1(MessageEditor):
          ...
     TESTSTEP UC1.2(MessagePreview):
          ...
     TESTSTEP UC1.3(MessagePreview):
          ...
     TESTSTEP UC1.4(SendingInfo):
          ...
```

Figure 6. Basic structure of a test case in TDL.

3.1. Assertions

This section describes how to check the values of components in a test case.

3.1.1. Basic assertions

Basic assertions are assertions put on basic components. Some components can contain only a single value (e.g., edit boxes). For such components, assertions look like:

```
CHECK ComponentName=Value
```

where the `ComponentName` is the name specified in ScreenSpec.

The basic form of a value is an *explicit string* (in double quotation marks). It is used to check whether the component has exactly the same value as specified (e.g. `CHECK Name="Olek"'` checks if `"Olek"` was typed into component `Name`).

One can also use *simplified regular expression*—a string (in double quotation marks) with wild card characters; e.g., `"Ole*"` or `"Ole?"`. Character `"*"` represents any string (with 0 or more characters), and character `"?"` represents any single character. The component value needs to match the string.

When more-advanced tests are required, *regular expressions* can be used (the syntax of regular expressions in TDL was borrowed from the AWK language [5], it assumes that each regular expression is enclosed in slashes). For non-expert users, this notation is rather hard to use and maintain, but it can facilitate the work of testers. For example, one can combine a number of similar tests by determining a regular expression on the basis of the available examples [2].

Table 2 summarizes the meaning of CHECK assertions for basic components.

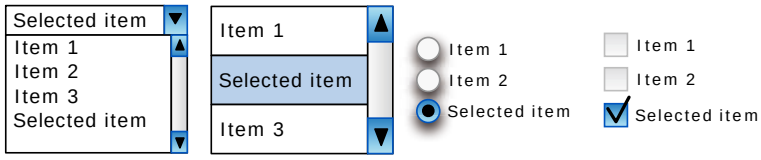
3.1.2. Semi-compound assertions

Semi-compound components (Fig. 7) can hold many values, some of them can be selected. For example, a combo box can have many options from which to choose, and one of them can be chosen by default. A list box can also have many options, and many of them can be chosen.

Table 2

Meaning of the CHECK assertion for basic components.

Component type	The meaning of the CHECK assertion
`BUTTON`	Check if the **caption of the button** matches the value.
`LINK`	Check if the **title of the link** matches the value.
`IMAGE`	Check if the **caption of the image** matches the value.
`STATIC_TEXT, DYNAMIC_TEXT`	Check if the **actual text** matches the value.
`EDIT_BOX, PASSWORD, TEXT_AREA`	Check if the **actual text** matches the value.
`CHECK_BOX`	Checks the checked status (possible values are: `CHECKED` and `NOT CHECKED`)



(a) A combo box (b) A list box (c) Radio buttons (d) Check boxes

Figure 7. Examples of semi-compound components.

An assertion for a semi-compound component is a compound sentence, which is a conjunction of predicates. The assertion starts with the `CHECK` keyword followed by the component name. The following lines (indented) describe the predicates. Predicates can be constructed using relational operators: `<`, `>`, `=`, `<=`, `>=`, `<>` (not equal) and the following functions:

- `POS(value)` – returns the lowest position of an option which matches the given `value`. Returns `-1` if none such option can be found. Positions are numbered starting with `0`.
- `COUNT(value)` – returns the number of options which match the `value`.
- `COUNT` – returns the number of all options in the component.
- `VAL(n)` – where `n` is an expression that evaluates to an integer number. Returns the value of n-th option.
- `SORTED` – returns `true` if the options are sorted ascending, `SORTED DESC` – returns `true` when options are sorted descending.
- `SELECTED(value)` – returns `true` if all options that match the `value` are initially selected.

The `value` in the presented functions is stated in the same way as for basic components.

Figure 8 presents an example of semi-compound assertion for a combo box.

Table 3 summarises the meaning of CHECK assertions for multiple-value components.

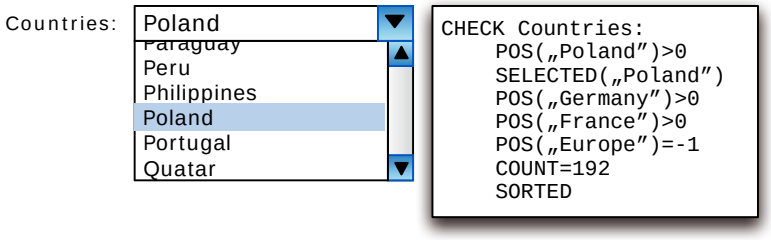


Figure 8. An assertion for a semi-structured component.

Table 3

Meaning of the CHECK assertion for semi-structured components.

Component type	The meaning of the CHECK assertion
COMBO_BOX, RADIO_BUTTONS	Checks **values of component options**. A single selected option is allowed, which represents the option that should be selected by default.
LIST_BOX, CHECK_BOXES	Checks **values of component options**. Many selected option are allowed, which represent the options that should be selected by default.

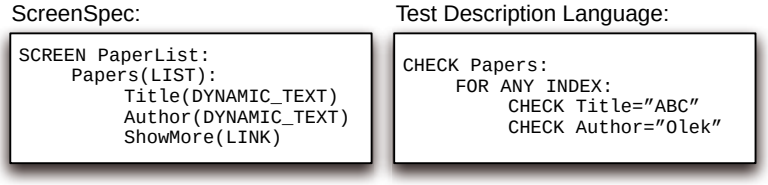


Figure 9. A ScreenSpec specification (left) and a corresponding structural assertion (right).

3.1.3. Compound assertions

Compound assertions are used to check values of compound components, such as tables or lists. Compound components represent a sequence of elements built from other components; each element structure is described in ScreenSpec. Therefore, compound assertions describe the expected values of components that belong to the elements. For instance, for a screen that presents a list of academic papers, we could write an assertion that, in the first place of such a list, there is a paper with a specific title.

Tables or lists in real-web applications usually contain many elements. From a testing point of view, we are usually interested in checking some elements, but rarely all of them. Therefore, requiring a tester to specify all elements explicitly would be a waste of time. TDL introduces quantifiers to make describing compound assertions easier.

Every compound assertion starts with the `CHECK` keyword followed by a compound component name (see Fig. 9). The following lines (indented) describe assertions for elements of the compound component. They start with a quantifier specification (e.g. `FOR ANY INDEX`), and the subsequent lines describe assertions for specific components that belong to the compound component.

The example from Figure 9 assures that the `Papers` list will contain at least one (`FOR ANY INDEX`) element, which `Title` equals "`ABC`" and `Author` equals "`Olek`".

There are three types of quantifiers in TDL: `EVERY` (universal quantifier), `ANY` (existential quantifier) and `ONE` (exactly one). There is also a quasi-quantifier `INDEX=n`:

- `FOR INDEX = n` – asserts that the element described by indented assertions will be at the specified position `n` in the compound component.
- `FOR ONE INDEX` – asserts that there will be exactly one element in the compound component that satisfies the indented assertions.
- `FOR ANY INDEX` – asserts that there will be at least one element in the compound component that satisfies the indented assertions.
- `FOR EVERY INDEX` – asserts that every element of the compound component will satisfy the indented assertions.

3.2. User actions

Test Description Language allows us to write user actions, which will be executed on the target system during system testing. Various components allow us to act on them differently. Table 4 summarizes all possible actions for ScreenSpec components. The *`Label`* argument will be replaced with the actual component label in a TDL script and *`Value`* with actual value.

Figure 10 presents a simple example of a screen containing a range of basic components and exemplary user actions.

```
SCREEN RegistrationForm:
     FormNumber(DYNAMIC_TEXT)
     Instructions(STATIC_TEXT)
     Username(EDIT_BOX)
     Password(PASSWORD)
     Description(TEXT_AREA)
     Private(CHECK_BOX)
     Country(COMBO_BOX)
     Sex(RADIO_BUTTONS)
     Newsletters(LIST_BOX)
     Interests(CHECK_BOXES)
```

(a)

```
SET Username="Lukasz"
SET Password="Pass"
SET Description="Description"
SET CHECKED Private
SELECT Country="PL"
SELECT Sex="Male"
SELECT Newsletters("General", "News")
SET CHECKED Interests("IT", "Dance")
```

(b)

Figure 10. A sample ScreenSpec specification (a) used to illustrate examples of user actions in TDL (b).

Table 4
User actions available for different components.

Component declaration in ScreenSpec	User action in TDL	Meaning of the action
Label (BUTTON) *Label* (LINK) *Label* (IMAGE)	CLICK *Label*	Clicks the component.
Label (STATIC_TEXT) *Label* (DYNAMIC_TEXT)	No action possible	Read-only components.
Label (EDIT_BOX) *Label* (PASSWORD) *Label* (TEXT_AREA)	SET *Label*=*Value*	Sets the value of the component.
Label (CHECK_BOX)	SET CHECKED *Label* SET UNCHECKED *Label*	Sets the state of the check box.
Label (RADIO_BUTTON)	SELECT *Label*	Selects the radio button.
Label (RADIO_BUTTONS) *Label* (COMBO_BOX)	SELECT *Label*=*Value*	Selects the option of the component.
Label (LIST_BOX)	SELECT *Label*(*Value1, ...*)	Selects values for the component. Many values are accepted.
Label (CHECK_BOXES)	SET CHECKED *Label*(*Value1, ...*) SET UNCHECKED *Label*(*Value1, ...*)	Checks the check boxes that match given values. Many values are possible.

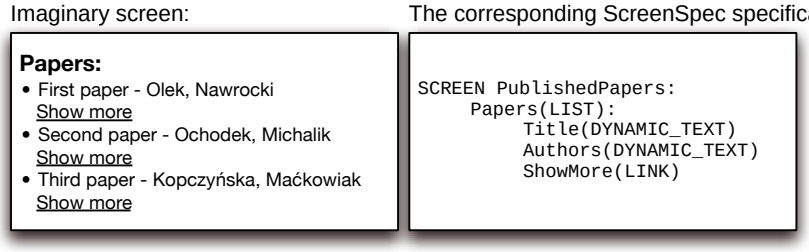


Figure 11. A screen presenting a list of published papers (left), and a corresponding ScreenSpec specification (right).

3.3. Labels and pointers

It is a bit more difficult to describe user actions for compound components because such components can have many elements, and we need to know exactly for which element a user action should be executed. For example, when a system displays a list of published papers (Fig. 11), for each paper there is a separate "Show more" link. How to choose the desired one? In order to do this, a pointer mechanism was introduced. Pointers can be declared for compound components – every list element or table row can be tagged with a label and pointed further in the TDL script. For example, in Figure 11, there can be three pointers declared for the "Papers" list: first for the "First paper", second for the "Second paper", and third for the "Third paper". Then,

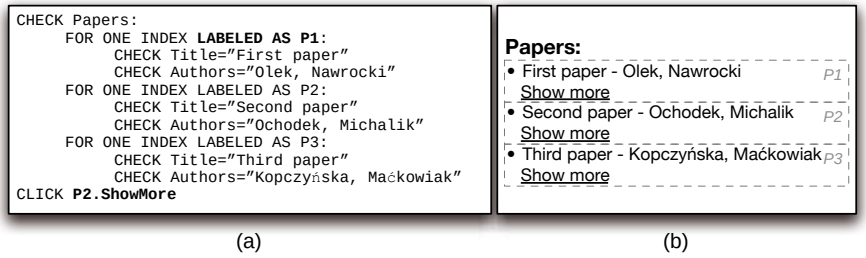


Figure 12. An example of a TDL script that uses labels to point a specific element on the list (a), and a corresponding screen (b).

when we would like to invoke a `CLICK` user action on the `ShowMore` component from the second element of the "Papers" list, we would prefix the label of the component with a pointer.

In order to enable declaring user actions for compound components, we need to have mechanisms for: 1) tagging elements of compound components with labels; and 2) pointing to them, when invoking user actions.

In order to point to a specific element, it is proposed to utilize the quantifiers described earlier. Two of them, `FOR INDEX=`*n* and `FOR ONE INDEX`, are able to choose one and only one element of a compound component. These elements can be tagged with labels. Declaring a label is done by appending the quantifier with keyword `LABELED AS` followed by a label:

- `FOR INDEX=`*n* `LABELED AS` *LabelName*, or
- `FOR ONE INDEX LABELED AS` *LabelName*.

Both quantifiers assert that the target screen will contain one and only one element of a compound component that satisfies the assertions declared inside them. Such an element can be referenced further by the label *LabelName*.

When labels are specified, invoking user actions is quite easy. For the purpose of TDL, the label of such a component is preceded with the label of an element and a dot. For instance, let as assume that a label `P2` is declared and it points to an element that contains a button `Submit`. We can invoke a click action on this component by writing: `CLICK P2.Submit`.

A complete example presenting labels and their usage is shown in Figure 12. The screen is expected to have a list of papers, containing three elements: "First paper", "Second paper", and "Third paper". For these elements, corresponding pointers are declared in TDL, labeled: `P1`, `P2`, and `P3`. Then, a user action is declared that performs a click on the `ShowMore` link from the element pointed with the `P2` label.

3.4. TDL and final web-applications

TDL can be used not only with mockups, but also with real web applications. To run a test, each web page needs to have elements with attributes `id` identical to the

labels used in the mockup design. In the case of `id` being used by a different web element, or being unavailable (e.g., due to a naming convention), a tester can provide a mapping between labels (used in mockups) and XPaths, which points to an element on a tested web page. Here is a basic example:

```
{
        "MessageEditor": {
                "Subject": "//input[@id='SubjectInput']",
                "Content": "//input[@id='ContentInput']",
        }
}
```

The above mapping uses JSON notation. It consists of an associative array with the name of a screen as a key (i.e., `MessageEditor`). As the value, there is another associative array. This time, a label of a component is used as a key (e.g., `Subject`) and XPath is used as a value (e.g., `//input[@id='SubjectInput']`).

4. Test case preconditions

During the design and execution of test cases, one assumes that the system is in a specific state. In TDL, required states can be achieved using preconditions. This section describes how they can be defined and used.

4.1. Describing preconditions with natural language

This approach assumes the user describes preconditions using natural language and a tester translates them into scripts (which will load the data into a system). Natural language is well understood by both users and testers. It allows us to describe the state in a vague way, taking a burden of detailed specification off of the user. For example, one can specify *The store contains 10,000 books, and among them, there is a book of "Jerzy Nawrocki"*, without the need to describe details of the other 9999 books. However, this lack of precision can be misleading, and scripts can lack important data.

Examples of a state description and a corresponding script are shown on Figure 13. At the early stages of a project, one can only use natural language (Fig. 13a). But after the design and implementation of a database, the description can be transformed into a script in SQL (Fig. 13b) or another language.

4.2. Describing Preconditions with FIT Tables

An alternative approach can be proposed using the FIT framework. A user can specify a state by preparing a set of tables, each table for each type of business object. The name of the fixture would determine the business object, and separate columns would describe its attributes. The fixture (see Section 1) should read these values and create objects in the system. A simple example is presented in Figure 14.

It is also possible to use FIT tables without using the FIT framework – a tester can write a software that reads FIT tables and put data into a database.

There is an author named "Jerzy Nawrocki", and he has three publications: "Pair Programming vs. Side-by-Side Programming"; "Balancing Agility and Discipline with XPrince"; and "Describing Business Processes with Use Cases".

(a)

```
DELETE * FROM authors;
DELETE * FROM publications;
INSERT INTO authors VALUES ('JN', 'Jerzy', 'Nawrocki');
INSERT INTO publications VALUES ('JN','Pair Programming vs. Side-by-Side Programming');
INSERT INTO publications VALUES ('JN','Balancing Agility and Discipline with XPrince');
INSERT INTO publications VALUES ('JN','Describing Business Processes with Use Cases');
```

(b)

Figure 13. A description of an initial system state in natural language (a) and a corresponding SQL script created by a tester (b).

Authors:

ID	First name	Last name
JN	Jerzy	Nawrocki

Publications:

Author ID	Title
JN	Pair Programming vs. Side-by-Side Programming
JN	Balancing Agility and Discipline with XPrince
JN	Describing Business Process with Use Cases

Figure 14. A specification of an initial system state by using FIT tables.

4.3. Usage of preconditions

To put a system into the required state (before executing a test case), one can use other test cases or execute scripts which, for example, encode the preconditions described in natural language.

In the first approach, all dependable test cases need to be listed after the keyword `PRECONDITON`; e.g., `PRECONDITON TC1, TC4` (see Fig. 5).

The second approach introduces three new actions:

- `OPEN` *`Label`* – open a screen with a given *`Label`* or a web page whose address is in *`Label`*;
- `RUNSCRIPT` *`Label`* – run a script called *`Label`* (e.g. this may be a file with commands in SQL or another language);
- `RUNTEST` *`Tag`* – run a test case marked with a *`Tag`*.

Figure 15 presents an exemplary precondition that may be used while testing.

One can use the keyword `SETUP` instead of `PRECONDITION`, which seems to be a more-natural keyword for programmers who are familiar with unit tests.

```
PRECONDITION:
        RUNSCRIPT CleanDataBase
        OPEN StartView
        RUNTEST CreateAccount
```

Figure 15. An example of preconditions in a TDL script.

5. Early evaluation

To check whether TDL is easy to use and maintain by IT-laymen, a small experiment was performed.

Two surveys were prepared: one each for TDL and Selenium (we chose this tool because of its popularity). Each survey had 2 tasks. In the first task, participants were asked to explain the meaning of 5 keywords on the basis of a use case written in FUSE notation, screen shots, and a test case in a given notation (the following keywords were used for TDL: `OPEN`, `STEP`, `CLICK`, `CHECK` and `SET`; and for Selenium: `get`, `find_element_by_id`, `click`, `getAttribute` and `send_keys`). The presented use case described a situation in which the actor logs into a web page and selects a layout (either a new graphical version of the service or an old one). In the second task, participants were asked to create a test case on the basis of a use case (adding a new comment on a web page) and a set of screen shots.

We are aware that tests in Selenium can be stored in an HTML file; however, using functions (like `get`) seems to be more popular among testers. In order to reduce complexity in the exemplary tests in Selenium, we decided to call function one after another without using a temporary variable to store a web element. Specifically, the notation `find_element_by_id("Login").click()` was used instead of the following version: `e = find_element_by_id("Login"); e.click()`. When the parameter passed to `find_element_by_id` was incorrect but the usage of the next function was correct (e.g. `click`), we treated it as a lack of understanding of the first function and an understanding of the second function.

There were 14 participants, all in their first year of study in a university computer science program. All of them declared to be unfamiliar with use cases and the testing of web pages. Participants were randomly assigned to two groups of seven participants: T that was asked to assess TDL and S that was asked to assess Selenium.

The average time in group T was 16 min, while in S it was 20 min. In group T, 68.57% of the keywords were understood correctly (i.e., participants described their meaning correctly and used them without any mistakes), while in group S, it was only 42.86% (see Table 5).

In the case of TDL, a common mistake was an incorrect reference to a step in a use case (only two participants noticed this dependency). Participants in S groups often forgot to load a page using `get` function. An incorrect `id` for function

Table 5
Results from the early evaluation (abbreviations: NCA – number of correct answers, PCA – percentage of correct answers).

		Keyword					Average
		`OPEN`	`STEP`	`CLICK`	`CHECK`	`SET`	
TDL	NCA	6	2	7	3	6	
	PCA	85.7%	28.6%	100.0%	42.9%	85.7%	68.57%
		`get`	`find`	`click`	`getAttr`	`send_keys`	
Selenium	NCA	2	4	5	2	2	
	PCA	28.6%	57.1%	71.4%	28.6%	28.6%	42.86%

`find_element_by_id` was also provided (an exemplary test case for Selenium used the same notation as in TDL – the label of a component and its `id` were similar).

It is very interesting that one of the participant in S group tried to add references to screens in tests.

6. Conclusions

This article presents the Test Description Language – a simple language that allows us to create acceptance tests of web applications with a link between use cases (in FUSE) and mockups (in ScreenSpec).

As was mentioned in the beginning of the article, this is only a small part of a bigger work, whose purpose is to allow for the creation of test cases for web applications in their early stage of development (i.e., when only requirements and low-fidelity prototypes of the layout are available).

The aim of TDL is to store test cases in a form that is easy to create and maintain by customers who are IT-laymen. We detailed how a test case can be created using this language, including verification points and user actions. We presented how to put a system into a required state by using preconditions. Moreover, we described an early experiment concerning the usage of TDL by IT-laymen, the results of which are promising.

Acknowledgements

We would like to thank to the anonymous reviewers for helpful comments that allowed us to improve the paper.

This work has been partially supported by the Polish National Science Center based on the decisions DEC-2011/03/N/ST6/03016.

References

[1] Framework for Integrated Test (FIT). URL `http://fit.c2.com/`.

[2] Bartoli A., Davanzo G., De Lorenzo A., Medvet E., Sorio E.: Automatic Synthesis of Regular Expressions from Examples. *Computer*, 2013.

[3] Beck K., Fowler M.: *Planning Extreme Programming.* Addison-Wesley, 2000.

[4] Burns D.: *Selenium 2 Testing Tools: Beginner's Guide.* Packt Publishing Ltd, 2012.

[5] Close D., Robbins A., Rubin P., Stallman R., van Oostrum P.: *The AWK Manual*, 1995.

[6] Constantine L. L., Lockwood L. A. D.: *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999. ISBN 0-201-92478-1.

[7] Haugset B., Hanssen G.K.: The Home Ground of Automated Acceptance Testing: Mature Use of FitNesse. *Agile Conference (AGILE), 2011*, pp. 97–106. IEEE, 2011.

[8] ISO: *ISO/IEC/IEEE 24765:2010 – Systems and software engineering – Vocabulary.* International Organization for Standardization, Geneva, Switzerland, 2010.

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

[10] Martin R., Melnik G., Inc O.: Tests and Requirements, Requirements and Tests: A Möbius Strip. *Software, IEEE*, vol. 25(1), pp. 54–59, 2008.

[11] Melnik G.: *Empirical Analyses of Executable Acceptance Test Driven Development.* Ph.D. thesis, University of Calgary, 2007.

[12] Nawrocki J.R., Olek L.: *Use-Cases Engineering with UC Workbench.* In: K. Zielinski, T. Szmuc (eds.), Software Engineering: Evolution and Emerging Technologies, Frontiers in Artificial Intelligence and Applications, vol. 130, pp. 319–329. IOS Press, 2005. ISBN 978-1-58603-559-4.

[13] Olek L., Nawrocki J.R., Ochodek M.: *Enhancing Use Cases with Screen Designs.* In: Z. Huzar, R. Kocí, B. Meyer, B. Walter, J. Zendulka, eds., CEE-SET, Lecture Notes in Computer Science, vol. 4980, pp. 48–61. Springer, 2008. ISBN 978-3-642-22385-3.

[14] Pressman R.: *Software Engineering – A practitioners Approach.* McGraw-Hill, 2004.

[15] Rettig M.: Prototyping for tiny fingers. *Communication of the ACM*, vol. 37(4), pp. 21–27, 1994. ISSN 0001-0782.

[16] Roehm T., Tiarks R., Koschke R., Maalej W.: How Do Professional Developers Comprehend Software? In: *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 255–265. IEEE Press, Piscataway, NJ, USA, 2012. ISBN 978-1-4673-1067-3.

[17] Shore J.: FIT and User Interface.
URL `http://www.jamesshore.com/Blog/Fit-and-User-Interfaces.html`.

[18] Snyder C.: *Paper Prototyping: The Fast and Easy Way to Define and Refine User Interfaces*. Morgan Kaufmann Publishers, 2003.

[19] Sommerville Y., Sawyer P.: *Requirements Engineering. A Good Practice Guide*. Wiley and Sons, 1997.

[20] Wynne M., Hellesoy A.: *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2012.

Affiliations

Łukasz Olek
Poznań University of Technology, Institute of Computing Science, Poznań, Poland,
`Lukasz.Olek@cs.put.poznan.pl`

Bartosz Alchimowicz
Poznań University of Technology, Institute of Computing Science, Poznań, Poland,
`Bartosz.Alchimowicz@cs.put.poznan.pl`

Jerzy Nawrocki
Poznań University of Technology, Institute of Computing Science, Poznań, Poland,
`Jerzy.Nawrocki@put.poznan.pl`

Received: 1.10.2014
Revised: 4.10.2014
Accepted: 4.10.2014