User Acceptance Testing for Agile-developed Web-based applications: empowering customers through wikis and mind maps

I. Otaduy, O. Diaz

PII: S0164-1212(17)30002-X
DOI: 10.1016/j.jss.2017.01.002
Reference: JSS 9906

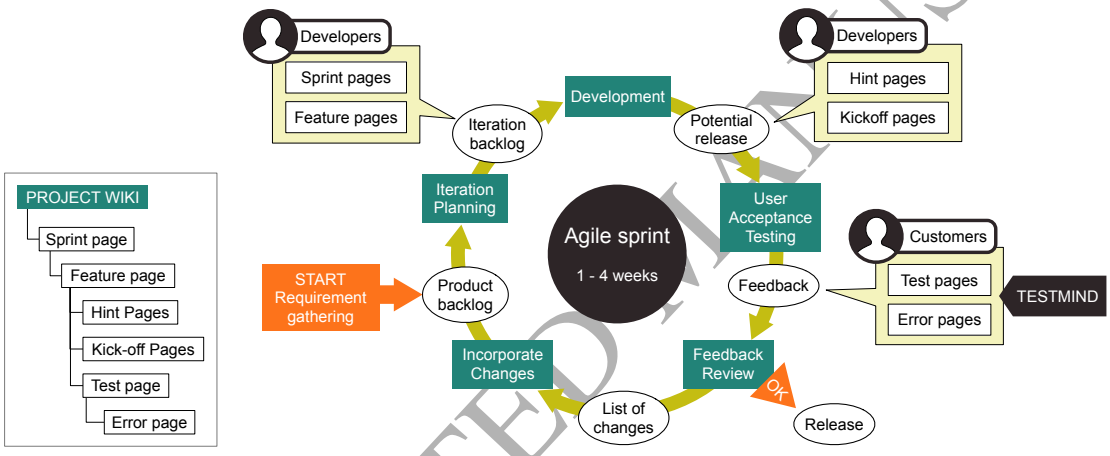
To appear in: *The Journal of Systems & Software*

Received date: 2 March 2016
Revised date: 3 January 2017
Accepted date: 5 January 2017

Please cite this article as: I. Otaduy, O. Diaz, User Acceptance Testing for Agile-developed Web-based applications: empowering customers through wikis and mind maps, *The Journal of Systems & Software* (2017), doi: 10.1016/j.jss.2017.01.002

Highlights

- A process to empower customers to conduct UAT in an Agile setting is proposed
- UAT is regarded as a collaborative endeavour between customers and developers
- Collaboration is supported through a Fitnesse wiki
- UAT is captured in terms of mind mapping
- Respondents valued asynchronicity, structuredness, and UAT documentation generation



User Acceptance Testing for Agile-developed Web-based applications: empowering customers through wikis and mind maps

I. Otaduy[a,*], O. Diaz[a]

[a] *ONEKIN Research Group. University of the Basque Country (UPV/EHU), P. Manuel Lardizabal 1, 20018 San Sebastián (Spain)*

Abstract

User Acceptance Testing (UAT) involves validating software in a real setting by the intended audience. The aim is not so much to check the defined requirements but to ensure that the software satisfies the customer's needs. Agile methodologies put stringent demands on UAT, if only for the frequency at which it needs to be conducted due to the iterative development of small product releases. In this setting, traditional in-person meetings might not scale up well. Complementary ways are needed to reduce the costs of developer-customer collaboration during UAT. This work introduces a wiki-based approach where customers and developers asynchronously collaborate: developers set the UAT scaffolding that will later shepherd customers when testing. To facilitate understanding, mind maps are used to represent UAT sessions. To facilitate engagement, a popular mind map editor, FreeMind, is turned into an editor for FitNesse, the wiki engine in which these ideas are borne out. The approach is evaluated through a case study involving three real customers. First evaluations are promising. Though at different levels of completeness, the three customers were able to complete a UAT. Customers valued asynchronicity, mind map structuredness, and the transparent generation of documentation out of the UAT session.

Keywords: Agile development, User Acceptance Testing, Test automation

1. Introduction

The software development process includes different test levels: unit testing, functional testing, integration testing or system testing. However, it is not until the customer uses the application that its adequacy to the client's needs is checked. This is referred to as User Acceptance Testing (UAT). During UAT, the software is tested in a real setting by the intended audience. Using the software itself as a means of communication and basis for discussion helps to make sure that as little as possible is misunderstood between the customer's expression of need and what actually gets built. Agile presents similar insights under the principle "customer collaboration over contract negotiation" [1]. Traditionally, this collaboration is realized as face-to-face meetings that tend to be held every 1 to 4 weeks to validate a runnable prototype. On the upside, this high frequency facilitates an earlier adjustment of expectations between developers and customers as for the final outcome. On the downside, it requires a larger customer involvement as compared to traditional methodologies where customer participation tends to be limited to the beginning and the end of the lifecycle. No wonder customers are not always ready to follow this quick pace. This makes developers complain about the lack of engagement and part-time dedication of customers [2]. So much so that the lack of customer engagement is regarded as a major stumbling block among Agile practitioners [3, 4, 5, 6]. Adverse consequences include "pressure to over-commit, problems in gathering and clarifying requirements, problems in prioritizing requirements, problems in securing feedback, loss of productivity, and in extreme cases, business loss" [4].

A common UAT approach is the presentation of software demos to customers [7]. This might raise three issues. First, in-person testing might entail agenda synchronization problems that exacerbates as the number of either attendees or places involved increase. Second, this method usually results in testing sessions becoming a training opportunity, and not a true test [8]. Finally, quite too often, customers offload their responsibility to the developers due to lack of resources, hence compromising the whole purpose of UAT. If the test scenarios are solely written by the development team (even if obtained out of meetings with customers), then there is a risk that meaningful scenarios are missed [9]. Hence, UAT should give customers the chance to interact with the software on their own, and find out if everything works as it should. Nonetheless, if customers are not tutored, they might not fully understand how the new software should be tested. The bottom line is that UAT excels when becoming a *truly collaborative* endeavour between developers and customers [2]. This is when wikis come into play.

Wikis are known to be beneficial in the workplace for

*Principal Corresponding author
Email addresses: itziar.otaduy@ehu.eus (I. Otaduy), oscar.diaz@ehu.eus (O. Diaz)

groups requiring a collaborative medium, with relatively small number of participants who are geographically distributed [10]. Testing is no exception. The notion of testing as a collaborative effort between developers and stakeholders, was pioneered by Ward Cunningham's *Fit* (Framework for Integrated Test), and its evolution, *FitNesse* [11]. This wiki platform facilitates team members to edit and execute tests. Before development, acceptance tests can be set to describe the expected behaviour of the software, generally expressed as an example or a usage scenario (e.g. using *Cucumber* [12]). After the development of a product release, UAT is performed to validate and accept the software. Unfortunately, the experience so far is for *FitNesse* to be still used as a mere test repository rather than a place where tests are collaboratively created [2]. Overcoming this situation requires customers to be able to work both on their own and at their own pace. This leads to our research question: **how to support self-paced UAT?**

We advocate for the use of mind maps as an accessible notation for describing UAT sessions (hereafter referred to as *"test maps"*). However, *conducting* UAT goes beyond *specifying* UAT. Conducting UAT involves assisting customers in coming up with their *test maps*. We need to elaborate on the tasks involved in UAT, from test setting to feedback documentation. In between, customers are assisted in different activities i.e. roaming, test coverage, enacting and commenting. The process should originate a *test map*. A video of a UAT session along this approach is available at `http://www.onekin.org/testmind`. This vision is evaluated through a case study involving three real web projects.

Contributions of this work are threefold. First, we propose a process that combines mind mapping and wikis to empower customers to conduct UAT on their own (i.e. self-paced testing). Second, we introduce a DSL for UAT whose concrete syntax is realized in terms of mind maps. Third, self-paced testing is realized using *FitNesse* as the wiki, and *FreeMind* as the mind map editor. These two platforms are bridged through *TestMind*, conceptually conceived as an editor for *FitNesse* that permits to capture UAT sessions as *test maps*. Self-paced testing is then devised as an iterative and collaborative effort where developers set the testing scaffold through the wiki, while customers build the test map that ends up as *FitNesse* pages.

The rest of the paper is structured along the main landmarks of the Design Science methodology [13]. Section 2 introduces the practice, i.e. UAT within Agile methodologies. Section 3 identifies the problem that arises within this practice, i.e., poor customer engagement. Section 4 draws six requirements to tackle this problem. Sections 5, 6 and 7 hold the main contributions of this work in terms of the process, notation and tooling to realize self-paced UAT. This approach is evaluated in Section 8 through a case study. Implementation details, related work and conclusions end the paper.

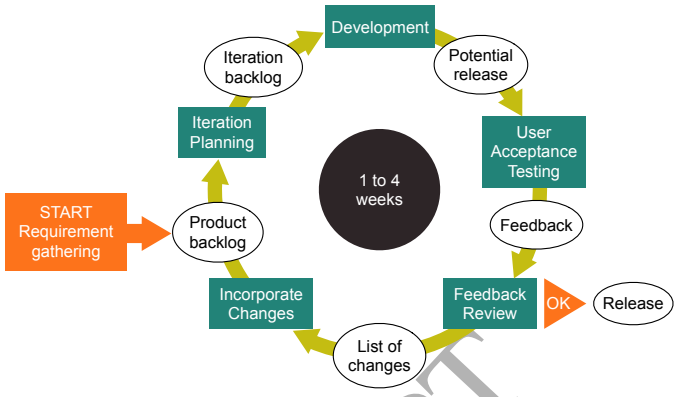


Figure 1: Process Outline.

2. The Practice: UAT in Agile processes

This work focuses on User Acceptance Testing (UAT) within Agile methodologies. These methodologies advocate for early delivery and continuous improvement through the iterative development of small sets of requirements in a short period of time, usually from 1 to 4 weeks (a.k.a *iterations*). After each iteration a runnable prototype of the final product is released and tested with real customers or a customer representative [7]. Figure 1 outlines this process. At the beginning, the features that compound the product are gathered into the *product backlog*. Next, the product development is carried out during a set of *iterations* or *sprints*. Each iteration planning includes selecting from the product backlog the subset of features to be developed (i.e., the *iteration backlog*). The development phase includes coding the new features and testing them through: unit testing (i.e., check small pieces of code), regression testing (i.e., ensure that changes have not introduced bugs into the previously developed modules) or system testing (i.e., determine whether the application meets the requirements as defined on the backlog). The output is a potentially shippable product that accounts for the features in the iteration backlog. This product is next tested for customer satisfaction during UAT. The feedback provided by customers is reviewed, and the changes requested are incorporated into the product backlog.

UAT aims to "replicate the anticipated real-life use of the product to ensure that what the consumer or end user receives is fully functional and meets their needs and expectations" [14]. It is important to note that UAT is not just acceptance testing, though UAT can take acceptance tests as a starting point. According to the Agile parlance, "an acceptance test is a formal description of the behaviour of a software product, generally expressed as an example or a usage scenario" [15]. For many Agile teams, acceptance tests are the main form of functional specification to capture business requirements. However, UAT occurs after a new product release has been developed and tested by developers. The underlying assumption is that new functionalities have already been tested against the initial requirements defined at the beginning of the project. UAT is

a new phase where customers check that the software satisfies their needs, regardless of whether these needs were previously defined or not. So "the core business is what is verified and validated and who better to do it than the business owners and the customers" [16].

3. The Problem

Ideally, UAT should be based on actual customer use, and performed by real customers. Unfortunately, the lack of customer involvement is being identified as a major jeopardy in Agile [2]. We searched the bibliography in order to detect the causes of this problem. Then, we grouped the causes along three main axes, namely: lack of time, lack of motivation and lack of knowledge (see Figure 2). However, this bibliographical review was not systematic. Hence, other issues might have been reported that were overlooked in our work. We didn't aim at conducting an exhaustive analysis of the rationales behind limited customer involvement, that would be a full-fledged research work on its own right in terms of a Systematic Literature Review. Here, we collect some evidences about the problem while our focus resides on attempting to solve it.

Lack of time. UAT testing goes beyond covering the defined requirements and lets customers freely wander along the application to achieve certain tasks. This implies customers to dedicate significant time away from their daily roles to participate [17]. However, getting stakeholders to invest their time on testing is not easy. As reported in [4], one developer complained that: "I've never worked on [a project] where the customer representative was given enough time to really be able to do the amount that they should." The literature points to three main causes: agenda issues, task issues and priority issues. The former is concerned with in-person meetings. The distance between the development team and their customers is an important factor leading to lack of customer involvement [4]. This situation aggravates as the number of people involved increases. The initial books about XP describe the on-site customer as a single person. However, the bias that a single customer representative can introduce has been documented in the literature as a cause of bringing about systems that will fail to satisfy many users [18]. Also, as stated in [7], "if management is so willing to assign to your project a supposed expert of the application domain, taking him or her away from tasks in that domain, you may wonder whether the person is really the most qualified". Beck also highlights that listening to just one person you risk of getting a system this person wants, but that won't be suitable for anyone else [19]. Today, there is a greater acknowledgement within the community that more than one customer is needed due to the large set of issues to be handled [20]. This seems also to be the conclusion in [21]: "I hear teams complain about product owners being insufficiently available and unable to make decisions on a short notice. Depending solely on the product owner appears to conflict with the team-based way the Agile process promotes". This makes the authors prefer the so-called product management teams. These teams "consist of multiple product managers, or product owners, which together are responsible for one or more products ... The main advantage of a product management team is it will have more knowledge than a single person. This provides better insight and support for the requirements towards the development team(s)". Yet, the more people intervene, the more difficult is to find a time slot that fits everyone's agenda. This problem exacerbates in globally distributed environments. Here, physical as well as temporal proximity between participants is even more challenging.

Another issue is the UAT being conducted manually. If the time is scarce, we should strive to find ways to assist this process as much as possible. A totally manual UAT approach can become tedious and time consuming which is being reported as discouraging for many customers [22, 23].

Lack of motivation. Any time is a lot if you are not committed to UAT. Shore et al. [24] state that developers should not underestimate the different tasks customers might need to cope with: set the appropriate priorities for the work, identify all the details that programmers will ask about, fit in time for customer reviews and testing, etc.This recognition of the customer's testing effort applies not only to the customer organization but also to developers themselves as reported by Martin et al.: "most of the programmers were unaware of the long hours the customers were working. This situation appears to be unsustainable, and so constitutes a great risk to agile projects, especially in long duration or high-pressure projects" [20]. This results in a lack of membership and less responsibility on the customers' part for the success or failure of the project [8]. Also, demotivated customers may not be aware of how to perform UAT or understand the importance of UAT, thus reducing the number of tests performed [25].

Lack of knowledge. Plenty of time and recognition do not yet guarantee good results. Customers might be used to traditional methodologies where UAT is conducted at the end of the lifecycle. This might bring an inertia when this *modus operandi* change in Agile processes [4]. Instead of the intensive participation demanded by Agile methodologies, they prefer to express requirements in traditional ways or in one-shot meetings [5]. Besides, making customers perform testing on their own might make them feel intimidated. For instance, in [26] a director testing the product confessed: "I don't know what to do. I don't know what to test and I don't know how to test." In some cases, the lack of technical knowledge lead to resort to QA personnel to help customers to develop their acceptance tests [27].

4. Meta-requirements for a solution

This section draws some meta-requirements extracted from the three limitations identified in the bibliography.

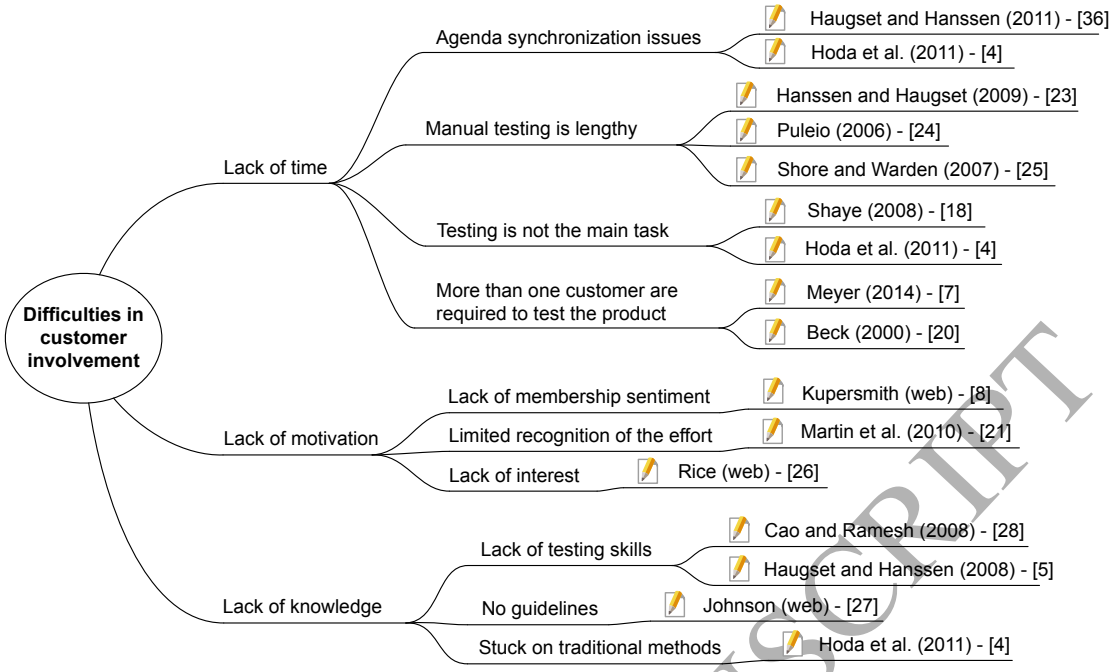


Figure 2: Root-cause analysis for the lack of customer involvement.

	Lack of Time	Lack of Motivation	Lack of Knowledge
MR1_async	✓	✓	
MR2_R&R	✓		✓
MR3_scaf			✓
MR4_maps			✓
MR5_mapping		✓	✓
MR6_feedback	✓	✓	
MR7_learnability	✓		✓

Table 1: Issues and addressing meta-requirements.

For each requirement, we look at existing solutions, if any. We ground the derived meta-requirements on research on Software Engineering. Table 1 maps the issues and their addressing meta-requirements.

Firstly, traditional face-to-face meetings constraint UAT to a defined time frame, and hence lessen the chances of the UAT participants to get distracted. However, some customers might not be able to spend two to four hours away from their desks for each Sprint. This sets our first meta-requirement:

> *MR1_async. Account for asynchronous collaboration through wikis*

Being a common place for knowledge management [10], wikis are known to be beneficial in the workplace for groups requiring a collaborative medium. This is a common scenario in software development [28] where wikis are proposed for improving software documentation [29], supporting collaborative requirements engineering [30], fostering software reuse [31] or sustaining acceptance tests [11].

In an Agile setting, wikis are proposed to support the application development lifecycle [32]. Story planning, backlog management, and acceptance testing find their way as wiki pages. A powerful exponent is *FitNesse* [11], a wiki platform geared to testing. This framework is based on Ward Cunningham's Fit (Framework for Integrated Test) [33], an open source framework that permits "to define acceptance tests as spreadsheets and then execute them using different languages". In *FitNesse,* Test pages capture tests as a collection of input-output pairs, navigation narratives or other forms. Test pages can be enacted by simply clicking a *"Test"* button at the top of the page. This testing framework is unique in using a wiki for creating and maintaining test cases. Here, the wiki becomes a repository for product requirements [34]. One of the drivers behind *FitNesse* is maintenance. Test automation projects often fail because tests are not designed for maintainability, and the overhead to keep the tests up to date becomes overwhelming. *FitNesse* helps to prevent this problem by using declarative though executable test specifications which are defined jointly with stakeholders.

Unfortunately, customers do not perceive *FitNesse* as an standalone communication and collaboration tool: offline discussions are conducted prior to changes made in the wiki [35]. This suggests that wiki asynchronicity is not enough to achieve self-paced UAT. This moves us to the second meta-requirement:

> *MR2_R&R: Offer customers a head-start using Record&Replay facilities*

UAT moves away from predefined scenario scripts to more exploratory testing, finding out about the application, what it does and what it does not. Record&Replay (R&R) tools permit to automatically generate browsing scripts as the customer freely navigates throughout the application. For instance, Selenium IDE offers this feature as a Firefox plug-in [36][1]. However, leaving customers to unrestrictedly wander throughout the application might end up in lengthy, unfocused scripts. This might be damaging in Agile. The short deadline for each Sprint makes it necessary to keep focus on the new developed features. This leads to our third meta-requirement:

> *MR3_scaf: nonintrusively guide customers during "exploratory" UAT through "testing scaffolding"*

If customers are not tutored, they might not fully understand how the new software should be tested. Developers should set "a scaffold" that assists customers during UAT. To this end, we propose the use of *Kickoffs* and *Hints*. Kickoffs set the web application under test (WAUT) at a state ready to be validated. This might imply setting up a database or running a browsing script in order to set the application at some specific point, saving the customer the intermediate navigation to reach the functionality to be checked. Notice that *Kickoffs* are not only a question of relieving customers from initializing the application but also of getting a state that might not be possible for the customer to achieve on his own due to e.g., lack of credentials (e.g. setting the company's holiday days).

However, customers are not professional testers. They might not be aware of good testing practices, e.g. checking not only the most common scenarios but also the faulty ones. Besides, demotivated customers might focus on getting things done with simplicity, without being too much involved in analyzing the different casuistry. This behaviour can jeopardize UAT as the chances of overlooking functionalities increase. To face this situation, developers can set *testing hints*, that is, short messages that aim at avoiding some aspects of the WAUT to go unnoticed. Hints are realized as notification messages that show up on the browser when the customer reaches specific application pages (see Section 5).

Scaffolding might help customers to keep focus. But once on focus, customers are left on their own. A notation is needed for customers to capture their UAT sessions. This moves us to the fourth meta-requirement:

> *MR4_maps: Capture UAT sessions as mind maps*

Different tools and languages have been proposed to permit the definition of acceptance tests. Toolkits such as *Cucumber* [12] or *FIT/Fitnesse* [33, 11] offer Domain Specific Languages (DSLs) to allow the description of so-called test-stories which can later be converted into executable test cases. These test cases can be used to measure and precisely describe the level of progress in the implementation of the requirements [37]. The use of DSLs to define test cases improves their readability, making them understandable to different stakeholders, from developers to business experts.

Based on this insight, this work investigates the use of a DSL for UAT based on mind maps. A mind map is a diagram to visually organize information (see an example in Figure 2). These maps are often created around a single concept, represented as a node in the center of a blank page, to which associated representations of ideas such as images or words are added [38]. Rationales for proposing the use of mind maps are twofold: familiarity and structuredness. As for the former, mind maps are a common notation among organizations for brainstorming and idea forming [39]. Chances are customers have already been exposed to this notation, hence facilitating adoption. In addition, mind maps bring an structure where ideas are radially disposed around a root node. For instance, *test maps* capture actions as leaf nodes of the map, while intermediate nodes serve to group actions based on the page where the action took place. This might well facilitate understanding in comparison with current *FitNesse* tables where actions are displayed as table rows, and no grouping exists.

But *conducting* UAT goes beyond UAT *specification*. Mind maps might provide a more familiar notation for customers. Yet, coming up with these *test maps* is not obvious. This results in the fifth meta-requirement:

> *MR5_mapping: Rephrase UAT as mind mapping*

UAT processes are rephrased in terms of mind mapping. In other words, common gestures in drawing a mind map (e.g. adding a child node) should now be overloaded with UAT semantics (e.g. test coverage).

Mind maps might well engage customers but the final consumers of this information are developers. For developers to make the best out of customers' UAT sessions, they should be able to reproduce them so as to ease the understanding of the test process specially in asynchronous collaborations. This brings us to the sixth meta-requirement:

> *MR6_feedback: Customers should be able to provide appropriate feedback documentation for developers to reproduce UAT sessions*

This poses stringent demands on feedback documentation quality which is at odds with the lack of both time and knowledge. Here, we investigate the extent to which this documentation can be automatically generated out of test maps. Specifically, *test map* structure helps obtaining test scripts that can be directly enacted from *FitNesse*. The aim is for developers to be able to replay the scenario conducted by the customer, and to reproduce the setting where potential mismatches arise.

[1] *Selenium* scripts can be enacted on *FitNesse* using the *Xebium* fixture. Available at `http://xebia.github.io/Xebium/`

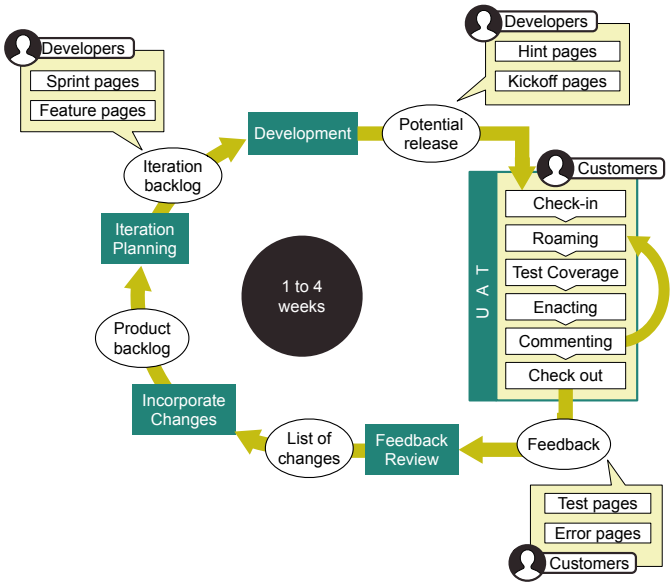


Figure 3: Agile process with a focus on UAT activities. Customers and developers collaborate through the wiki pages.

Even though the use of mind maps might ease the test definition task, the customer role tends to be transient, i.e. it is frequently played by different employees depending on the application at hand. That is, there might not be a second time for an employee to play the role of the customer in another application. Hence, the UAT learning effort should pay off for a single application. Therefore, the last meta-requirement consists on reducing the learning barrier for the UAT solution:

> *MR7_learnability. The solution should be intuitive enough for customers to be able to grasp it rapidly*

To conclude, we advocate for an asynchronous collaborative approach to UAT. The instantiation of this theory is then threefold: a process that describes how collaboration takes place (Section 5), a notation to capture UAT sessions (Section 6), and a tool that supports this collaboration (Section 7).

5. The process: iterative and collaborative UAT

Wikis are widely used in industry for teamwork and collaboration [10]. Wiki pages might respond to different purposes. For instance, Wikipedia holds Article pages but also Help pages, Template pages, etc. Different questions arise about: (1) *what* type of pages a UAT-aimed wiki would contain; (2) *when* are they created with respect to the Agile cycle, and *(3), who* creates them (i.e. developers vs. customers). Figure 3 outlines an answer by spreading out the UAT box. Next paragraphs delve into the details. A *Calendar* application is used as the running example.

Before Development. Developers first start by defining **Sprint pages**, i.e. pages that hold information about the Agile iterations, such as the list of features included in the sprint backlog. Each feature is fully described through a dedicated **Feature page**. These pages hold a list of the customers assigned to testing and a brief description of the feature to be developed. This information is later enriched with links to the Test pages where this feature is being tested out (see an example in Figure 4 (3)).

After Development. Once a new product release is available, developers set a scaffold to assist customers during UAT. To this end, we propose the use of **Kickoff pages** and **Hint pages**. Kickoffs set the web application under test (WAUT) at a state ready to be validated. As an example, consider the *Calendar* application. *Calendar* permits adding new events as well as modifying the existing ones. Rather than starting with an empty calendar, *Kickoffs* might set the calendar in a stage ready for UAT, e.g. calendar with events set for you, calendar with public holidays already set, calendar with events jointly arranged with your colleagues, etc. Figure 4 (2) displays the case of the Kickoff "Calendar with events set for you". Rather than starting with a empty calendar, this Kickoff initializes the calendar with some events before the UAT session starts. Kickoffs are realized as scripts, specifically a sequence of Selenium commands that achieve the desired initial state. Kickoff pages might provide a *more-information* hyperlink to extend the information (i.e. kickoff description, usage suggestions, screenshots of the point where the kickoff leads to, etc).

As for Hint pages, they aim at suggesting customers to perform some testing practices they might not be aware of (e.g., "attempt to introduce wrong values"), or getting their attention to some functionality aspects that might be overlooked (e.g., "try the different agenda views"). Hint pages describe testing hints as pairs *(context, message)* where *context* refers to the point in the browsing navigation where the message should pop up. For the sample case, two hints are defined (Figure 4 (1)): *"if the navigation reaches the URL action/index.php, then suggest the customer to check different agenda views"*, and *"if the page action/card.php?action=create is loaded, then inform about the possibility of using wrong input data"*. Implementation wise, this is supported through a Selenium IDE plug-in. The plug-in tracks the context. When the context is reached, the message is visualized in the browser's hint bar (see Figure 7 (2)). Notice, hints should not give any clue about *how* to achieve tasks as this is part of the UAT itself.

During UAT. It is now the turn for customers to provide **Test pages**. Test pages are not produced out of the blue but after iterating along the following tasks: *roaming* (i.e. freely wandering around the application), *test coverage* (i.e. adding different data sets), *enacting* (i.e. running the application with the provided data sets), and *commenting* (i.e. providing remarks about test outputs). This cycle is framed within *check-in,* where the UAT session is initialized, and *check-out*, where the UAT session is finally documented as a Test page on the wiki.

Figure 4: Wiki structure and distinct scaffolding pages for the *Calendar* application: *Hint* page (1), *Kickoff* page (2), and *Test* page (3). Test pages are enacted through the *"Test"* button (A).

A Test page contains: the testing setting (e.g., operating system, browser version, etc.), a script table and a decision table (see Figure 4(3)). Script tables hold a sequence of executable commands which stand for the test case. Clicking on the *"Test"* button automatically replays the script, i.e. a Firefox window pops up and the user interactions are reproduced. In this way, *FitNesse* permits developers to easily replay the scenario conducted by the customer. Script tables also hold the customer's comments risen during the testing. Comments are captured as note rows (e.g. *"When I first enter the agenda ..."*). In addition, *decision tables* hold data sets with which the test case should be checked out. If the test did not meet the customer expectations for a specific data set, then the corresponding row might hold a hyperlink to an **Error page**. Error pages document the script results: the customer expectations about the test outcome and the screenshots taken.

Figure 3 highlights this collaborative nature of UAT. Customers can facilitate *Test* pages at any time, provided the testing deadline is not over. All these *Test* pages are saved under the *Feature* page representing the requirement that is being tested. Once the testing deadline is over, developers can go back to *FitNesse* and replay the customers' *Test* pages. In this way, developers can make a more informed decision about whether the current development really meets customers' expectations. At this point, additional hints and Kickoffs can be defined, and the UAT deadline be extended for another UAT round. The bottom line is that UAT is not a one-shot effort but a collaborative and continuous endeavour between developers and customers.

However, this vision has a main stumbling block: test scripts are outside most customers' competences. The question is then

how can Test pages be created by customers?

This rises two main issues: a notation for capturing UAT sessions (Section 6), and assisting customers in coming up with test cases using this notation (Section 7).

6. The notation: test maps

The formalism used to capture tests very much depends on the kind of tests. In acceptance testing, *Cucumber* is a popular tool [12]. As another example, Selenium provides a different notation (called Selenese) that permits to define test cases in terms of a sequence of user interactions with the application UI [36]. No matter the notation used, an important requirement is for the test cases to be executable since users other than the author might need to run the test. However, manually defining test cases following a restricted notation implies some complexities from which customers should be sheltered. Here, we advocate for the use of mind maps. The insight is that mind maps might provide a balance between usability (i.e. visual and familiar notation) and formality (i.e. to be structured enough to be enactable).

Broadly, UAT sessions collect three main types of data: (1) information provided by developers to set up the customer testing (*scaffolding)*; (2) test case definitions generated by customers *(UAT Cases),* and (3), test case enactments (*UAT Scripts*). To capture these concerns, we resort to mind maps. Figure 5 depicts the different kind of nodes involved and their structural arrangement. Each node holds a pair *(icon, label).* The icon conveys the semantics, i.e. it indicates what the node stands for. The label names the concrete realization. The use of these nodes along this arrangement leads to a *test map.*

Therefore, *test maps* are mind maps, but not all mind maps are test maps. That is, *test maps* restrict the structure and kind of participating nodes. In short, *test maps* become the graphical representation of a Domain Specific Language (DSL) for UAT. Coming up with a DSL implies identifying the main concerns of the UAT domain, providing a metamodel that captures the main relationships (a.k.a. abstract syntax), and finally, facilitating a graphical notation to specify the DSL expressions (a.k.a. concrete syntax) [40]. This Section focuses on the expressiveness of test maps while other design considerations are left for Appendix A. Readers can follow the process of coming up with a *test map* in Figures 6, 8 and 9.

6.1. The expressiveness of test maps

For the purpose of this work, UAT sessions are framed by some information provided by developers (e.g., the list of features to test, the list of available *Kickoffs*) called the *Scaffolding.* The goal of UAT is to validate a backlog feature by defining a *UATCase* and checking out the result of different enactments (*UATScript*) using several input values. Next paragraphs delve into the distinct concerns.

Scaffolding. From a user's perspective, this refers to the Sprint features to be tested and the available kickoffs that set up the UAT scenario. Broadly, the feature to test becomes a node of the map from where its different kickoffs hang up (see Figure 5).

UAT Cases. Test cases are defined as "sets of test inputs, execution conditions, and expected results" [41]. In our approach, test cases represent a sequence of steps denoted as *"UATActions"*, i.e. commands that mimic a user interaction with the web interface (e.g. click, data input). To ease customer understanding, *UATActions* are grouped into the *Page* where these actions happen. As for the expected results, they are defined as test oracles, that is, methods for checking whether the WAUT has behaved correctly on a particular execution [42]. In scripting languages, these oracles are realized as assertions, i.e. checking actions to be performed in order to validate the UAT case result (e.g. check the content of an element, check that a specific page is loaded). Traditionally, assertions are boolean expressions set by developers at a specific point in a program which will be true unless there is a bug

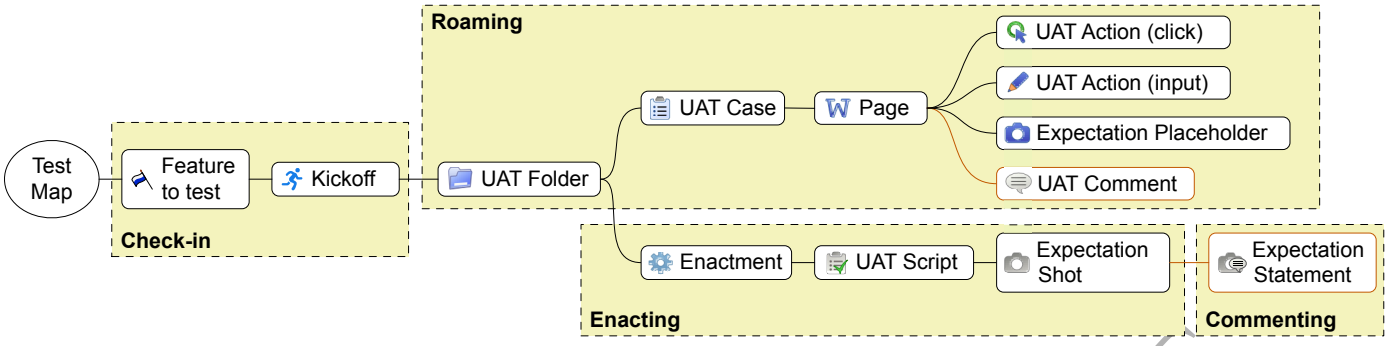


Figure 5: *Test maps*: mind maps to capture UAT concerns. Map nodes are overloaded with UAT significance. Their operational semantics is defined through a Domain Specific Language (see Appendix A).

in the application. Assertion definition commonly implies two concerns:

- *When*, which indicates the application state on which the checking should be performed. It is described in terms of the content of the application's variables, the page being rendered, etc (e.g., "when I fill out the form", "when I click this hyperlink").
- *What,* which stands for the condition to be satisfied (e.g., "when I fill out the form (when), the missing fields should be highlighted (what)", "when I click this hyperlink (when), a specific page should be loaded (what)").

UAT cases are test cases, and hence, they can be similarly described. The difference stems from both the aim (validation vs. verification) and the audience (customers vs. developers). Being a validation technique, we consider UAT to be more a confirmatory practice than an investigative practice, in the sense that the customer knows "what the good result is and is trying to find proof that the product conforms to that result" [43]. This basically means that the customer acts as the test oracle. We then rephrased the *When* and the *What* as follows,

- *When* refers to the specific points in the browsing navigation where the customer might have some expectations about the UI rendering (referred to as *ExpectationPlaceholders*),
- *What* holds the customer's comments about whether or not the UI met those expectations (referred to as *ExpectationStatements*).

While *ExpectationPlaceholders* are set during the *UATCase* definition, *ExpectationStatements* are captured after the test execution (see next). In addition, customers can include *UATComments* (e.g., *"The font is too small"*, *"The logo is not well positioned on this page"*) to transmit ideas arisen during the *UATCase* creation.

UAT Scripts. These scripts are the executable artifacts generated from *UATCases*. Each *UATScript* exercises a *UATCase* using different input *data sets* to check out the behaviour of the application in different scenarios. For each set of input data, the customer might have a different expectation (e.g., *"if I introduce 16/02/02, the application should pass"*, *"if I introduce, 16/42/42, the application should throw an error"*). As stated in the previous section, the customer manually defines this expectation based on application screenshots that are taken during the test execution (*ExpectationShots*). Once checked, *ExpectationStatements* can be created in order to point out whether the test passed (i.e., the application worked as desired) or failed (i.e., the application did not behave as expected).

7. The tool: Assisting users to come up with test maps

Notation plays a key role in facilitating user involvement. But notation alone will not succeed unless appropriate assistance is put in place. This moves usability at the forefront. ISO defines usability as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [41]. For our purpose, this can be rephrased as the ease with which *customers* can use *test maps* for *achieving UAT*. We understand "achieving UAT" as creating the final *Test* pages that will be shared on *FitNesse*. We then elaborate on the tasks that end up with a *Test page*. These tasks are listed in Table 2: check-in, roaming, test coverage, enacting, commenting and check-out. This section describes how these tasks are conducted through *TestMind*, a plug-in for the *FreeMind* mind-map editor. A video of the process is available at `http://www.onekin.org/testmind`.

FreeMind is a popular editor for mind mapping [44]. Figure 6 displays the main screen regions for this editor: the upper canvas to edit the mind map structure; the lower canvas to extend the information related to the node being selected (in the Figure the selected node is *"Calendar has evens set for you - Kickoff"*); the toolbar on the left for adding icons; and the toolbar on the top for managing the mind map.

Task	Description	GUI Gesture
Check-in	Setting the context (e.g. the backlog feature to be tested, the Kickoff to be used)	Click on *"check-in"* button
Roaming	Wandering around the application in a free-way	Tab on *Kickoff* node
Test Coverage	Working out different data sets	Bottom canvas editing with *UAT Case* node selected
Enacting	Running the application with different data set	Tab on *Enactment node*
Commenting	Providing remarks about application outputs	Tab on *ExpectationShot* node
Check-out	Collecting UAT session insights as a Test page	Click on "check-out" button

Table 2: Tasks involved during UAT.

TestMind customizes *FreeMind* for *test map* specification. Screen wise, the difference with the standard *FreeMind* stems from the two icons on the left of the toolbar (see Figure 6(1)). The *check-in* button initializes the definition of a *test map* by downloading from *FitNesse* the features to be tested by the current user. The *check-out* button saves the canvas content as a *Test* page in the same *FitNesse* installation. In between, customers need to elaborate the test map along the stages detailed in Table 2. On each stage different nodes are gradually added to finally obtain a complete *test map* (see Figure 5). Basically, customers' main tasks are threefold: creating the *UAT Case*, providing data sets, and confirming expectations after the execution of the test scripts. Moving from one stage to the other is achieved either via standard node creation in *FreeMind* (*TAB key* pressing[2]) or via selecting toolbar buttons (see Table 2). The rest of this section describes the details.

7.1. Check-in

It could have been possible for customers to directly access *FitNesse* to get informed. However, customer disorientation in accessing the wiki content and the lack of appropriate access control mechanisms put this option aside. We then explore an alternative where *FitNesse* is accessed from a mind mapping tool through a plug-in called *TestMind*.

At the time *TestMind* is installed, users are prompted for their credentials to access *FitNesse*. At *check-in* time, these credentials are used to connect to the *FitNesse* installation, and to recover UAT duties for *this* customer. Figure 6 shows the outcome for the customer *Waldo*. *Waldo* can see how his participation is required in the testing of two features: *EventModification* and *NewEventAddition* (Figure 6 (2)). Roughly, FreeMind's canvas depicts the mind map counterpart of the *FitNesse* pages set on the *Scaffolding*. The mapping between both representations is up to *TestMind*.

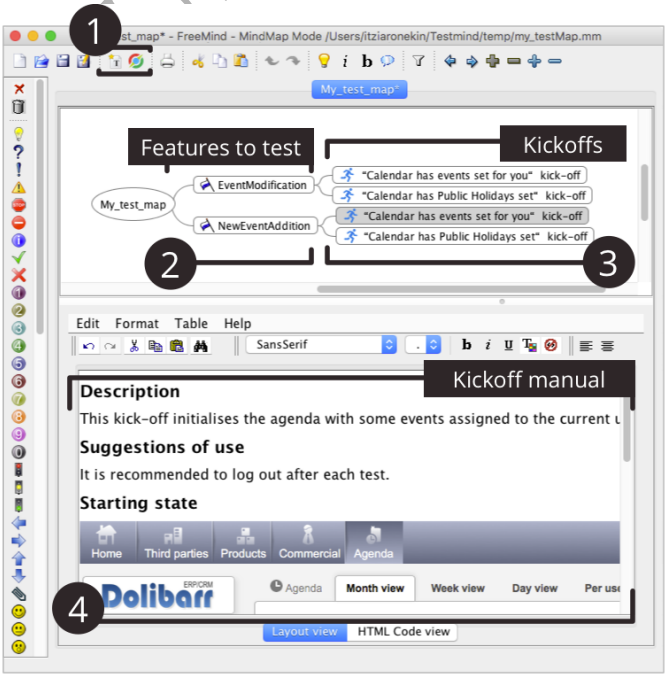


Figure 6: **Check-in.** Customer attention is drawn to two features: *EventModification* and *NewEventAddition* (2). For each feature, Kickoffs are displayed as sub-nodes (3). Select a *Kickoff* node for a description to pop up in the lower canvas (4).

[2]TestMind overrides FreeMind's default daemon for node creation. Based on the kind of node selected, the daemon invokes the corresponding task. For instance, the daemon associated with *Kickoff* nodes invokes Selenium IDE so as to be able to create a child *UAT Case* node. This is part of the TestMind plug-in functionality.

7.2. Roaming

Now, Waldo is ready to provide a new UATCase. He selects a Kickoff scenario (e.g. *"Calendar has events set for you"*), and uses the *FreeMind* gesture to create a new child node: TAB key press. This gesture moves the control from FreeMind to Firefox. A new window is opened, the Web application is loaded, and the customer is positioned at the Kickoff point. From then on, the customer can wander freely. Using Record&Replay tools (specifically, Selenium IDE[3]), customer actions start being recorded. Three enhancements are however introduced:

- comment as you go (see Figure 7 (1)). Customers can introduce comments as they browse. Comments are automatically accompanied with a screenshot of the current page. No need for the customer to manually take screenshots.
- testing hints (see Figure 7 (2)). Notifications can be triggered on the browser bar when the right context is reached. The aim: getting customer attention to good practices on testing or to application aspects that could go unnoticed. In the example, TestMind suggests the customer to try different calendar views.
- setting *ExpectationPlaceholders* (see Figure 7 (3)). These are points in the application that might rise some expectations as for their UI outcome. They are captured through the *camera* button. By clicking, the current navigation step is turned into an *ExpectationPlaceholder*. These placeholders are later used to present the user the screenshots taken when the application is replayed with different input data sets (see Subsection 7.4).

Clicking the *"Stop"* button ends the recording (see Figure 7 (4)). This closes Firefox, and passes the control back to *FreeMind* whose canvas now displays a new *UATCase* node (see Figure 8 (1)). This node is automatically generated after the recorded *Selenium* session: clicks, comments, and expectation placeholders find their way as mind map nodes.

The mind map now represents Waldo's performed interactions with the WAUT. Next, Waldo can select a node in the mind map and see its content in the lower canvas. The upper canvas and the lower canvas are in sync so that selecting a different node in the mind map makes the lower canvas to be accordingly refreshed. This is useful in two scenarios:

- to ease user orientation in UATCases. UATCases can comprise a large number of nodes. By including a screenshot of the page where each action was conducted the customer can better pinpoint where actions fit in the test by simply looking at the lower canvas,
- to manage test data. The input data sets are rendered as an HTML table on the lower canvas for *UATCase* nodes. In Figure 8 (2), the lower canvas summarizes the data being provided for the page at hand: *Title, Start date, aphour, End_dat, p2hour* and *event_assigned_to*. Column names are derived from web elements' available attributes, such as the id, the title or the label.

The readability of the *test map* very much depends on the web application. For example, if form fields are not correctly labeled, the corresponding node's text might not be sufficiently representative. This can reduce the testability of the WAUT[4].

The customer can create more *UATCase* nodes at will by simply pressing the TAB key on the *Kickoff* node of choice. Each test case is reflected as an additional child of the *Kickoff* node at hand.

7.3. Test Coverage

Data sets are traditionally held in Excel tables or CSV files. Next, testing programs map this data to the appropriate test function parameters. By contrast, TestMind resorts to *test maps* as *a situated way* to introduce additional data at the point this data is requested. Data is requested by input actions. *Action* nodes hang from *Page* nodes which, in turn, hang from *UATCase* nodes. Hence, *UATCase* nodes aggregate the data from their underlying *Page* nodes which, in turn, gather the data from their *Action* children. This tree-like structure allows to see the data set at different levels. Users can have a whole view of the data set by selecting a *UATCase* node (see Figure 8 (2)). Alternatively, if a *Page* node is selected, the bottom canvas will limit the display to the data used in this page.

As an example, consider a *UATCase* which visits pages P1, P2 and P3 which hold entry forms F1, F2 and F3, respectively. The customer can provide new data sets for forms F1, F2 and F3, by placing himself into the *UATCase* node. Alternatively, he might only focus on F1 by selecting the corresponding *Page* node (P1), and provide new data combinations just for this specific form. In this case, *TestMind* completes F2 and F3 with the default values (i.e. those obtained during recording). Figure 8 displays how the *UATCase* previously recorded is supplemented with additional data sets. In this way, customers can easily add new data by simply creating new rows on this table[5]. The

[3]Selenium IDE is an open source plug-in for Firefox that enables recording user actions and replaying them for testing purposes.

[4]ISO defines testability as "the effort required to test software"[41]. As HTML labels become identifiers for nodes in *test maps*, the testability of the application using *TestMind* greatly depends on whether HTML elements are meaningfully labelled or not. Hence, customers' understanding very much depends on developers creating an accessible application, where labels and ids are not randomly defined, but based on the HTML element's semantics. In general, W3C WCAG 2.0 recommendations should be followed [45].

[5]Consistency is maintained among the different aggregation views so that data inserted in a *Page* node propagates both upwards (to its *UATCase* node) and backwards (to its *Action* node).

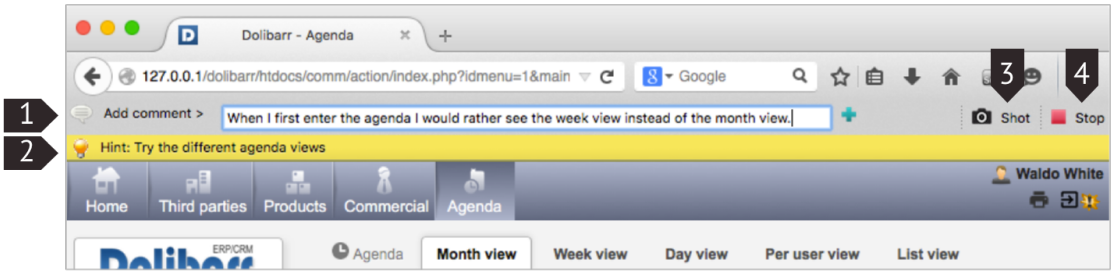


Figure 7: **Roaming**. Obtaining *UATCases* through Selenium IDE. Enhancements to this IDE include: (1) comment bar; (2) hint bar; (3) expectation-placeholder bookmark; (4) stop recording.

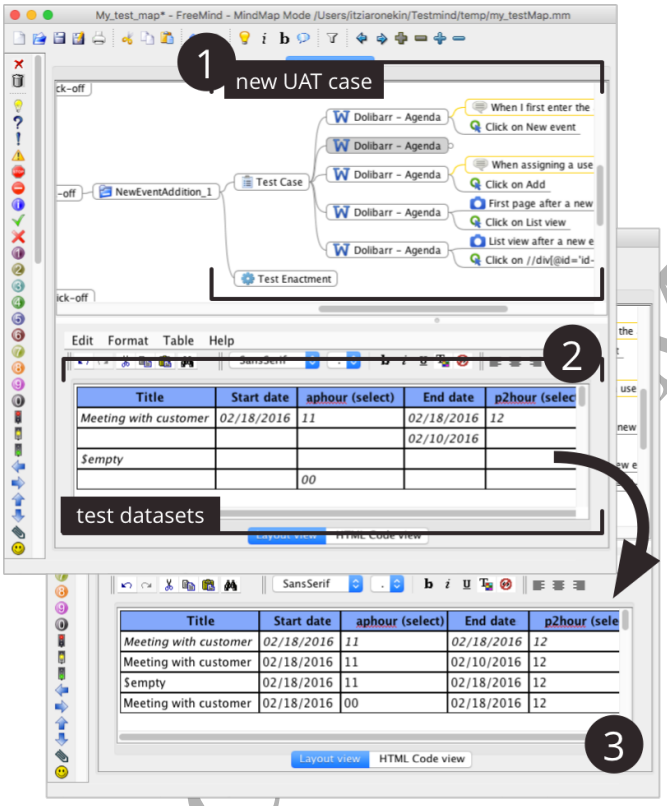


Figure 8: **Test Coverage.** When a page node is selected on the test map, the lower canvas shows a table where new input data sets can be introduced. Here three scenarios are added (2). *TestMind* complements the empty cells as follows: blanks are replaced by default values (taken from the first row) while the *$empty* keyword is used to deliberately leave a blank value. The resulting table is depicted in (3).

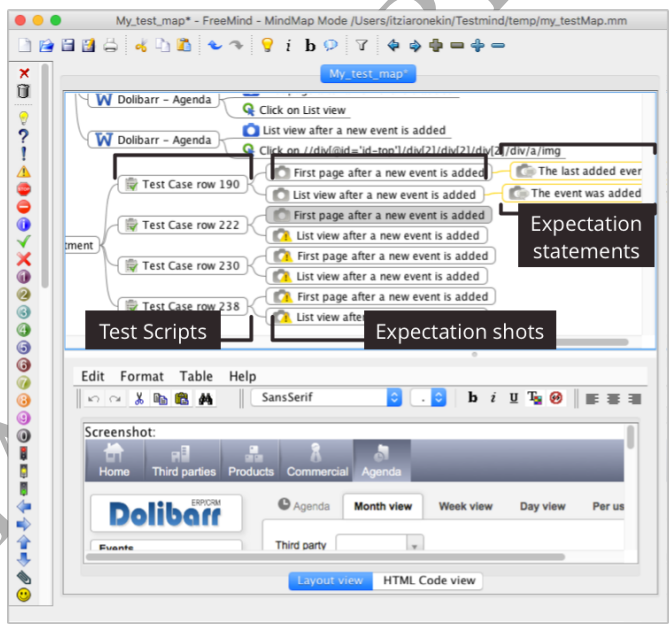


Figure 9: **Commenting**. Customer adds two *ExpectationStatement* nodes for *Test Case row 190*.

expectations are for this situated data provision (i.e. providing test data at the place where the data is going to be consumed) to facilitate customers' understanding and orientation.

7.4. Enacting

Once new input data sets are provided, the customer positions himself on the *Enactment* node of the corresponding *UATCase* and presses the TAB key. Behind the curtains, *TestMind* generates *UATScripts*, enacts so-generated scripts, and adds the outcome as map nodes for the customer to check. Figure 9 shows the output for the data coverage in the previous subsection: four data rows give raise to four *UATScript* nodes. In addition, the test map is enriched with the *ExpectationShot*s taken at the ExpectationPlaceholders defined by the customer while browsing the WAUT.

7.5. Commenting

At the end of the *enacting* stage, the test map depicts a *UATScript* node for each input data row (e.g. *Test case*

row 190). The customer can now go through the different *ExpectationShots*, inspect them, and report whether his expectation is fulfilled or not (i.e. add an *ExpectationStatement* node). By now, the customer might have a better understanding about the application. He can decide to go back to provide additional test data, or even try a new wandering in order to create new *UATCases*. Once finished, the UAT session is to be exported to FitNesse. This moves us to the *check out*.

7.6. Check out

Test maps are stored as *FitNesse* pages. Each *UATCase* node results into a *Test* page. A Test page contains: the testing setting (e.g., operating system, browser version, etc.), a script table and a decision table (see Figure 4 (3)). In short, *TestMind* relieves customers from the burden of generating feedback documentation. No more need to write emails or text documents, freeing up time to explore different task flows and input data set combinations.

8. Evaluation

This Section looks at whether *TestMind* is an effective process and tool for self-paced UAT. We believe UAT is not only a technical but also a social issue. Limited customer involvement is frequently the result of poor motivation and limited membership sentiment. This mixture of technical and social issues advocates for the use of a case study to evaluate *TestMind*.

Case studies in software engineering are "an empirical enquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified" [13]. A case study differs from a laboratory experiment in its focus, based on depth and context [13]. A laboratory experiment reduces complexity by controlling, even eliminating, factors that can interfere with the experimental results. In contrast, complexity is essential for a successful case study, as it investigates multiple factors, events, and relationships that occur in a real world setting. We are interested in measuring *TestMind* effectiveness for conducting UAT, together with the reactions it produces on its environment: how the customer's colleagues react, the level of interruptions that arise while testing, the level of customer procrastination, and, finally, whether the customer's perception about testing changes after using *TestMind*. UAT is predominantly about the customer. Developers play an ancillary role as the definers of the UAT scaffolding. Hence, this evaluation focuses on the customer, leaving for future work the study of the developers' perspective.

A case-study design includes the methodology (subsection 8.1) and the subject selection (subsection 8.2). The rest of the subsections introduce the research questions and the results. Results were gathered using different data collection methods, namely subject observations, interviews and questionnaires [46].

8.1. Methodology

Testing sessions were conducted at the customers' places and supervised by one researcher who played both the roles of facilitator and observer. A *FitNesse* installation was prepared for each of the customers' web applications: sprints, features, kickoffs and hints were accordingly defined. At the beginning of the evaluation session, the customer was informed about the objectives of the test and she was interviewed about her previous testing experiences, putting a special focus on detecting the problems or limitations so far. A first questionnaire was then filled out by the subjects, so as to gather demographic information and basic data about their technological background. Then, customers were introduced to *TestMind* with a basic explanation of about 30'.

After this introduction, the control over *TestMind* was handed over to the customer. The researcher accompanied customers along the usage of *TestMind*, observing their actions, solving questions and taking notes. Also, we encouraged users to express their sensations and impressions during the evaluation, based on the guide on Appendix C. Once the testing session was finished, participants were asked to fill in a questionnaire without the monitorization of the researcher. This questionnaire rates different aspects of *TestMind* through Likert scales (see Figure 10).

Notes about customer sensations were taken by the researcher, along with data about the number of tests correctly generated and their complexity (number of comments and *ExpectationPlaceholders* added). For result triangulation, the data gathered during the evaluation was verified against the answers given on the Likert questionnaire [46]. Each interview lasted for about one hour.

8.2. Evaluation subjects

TestMind was evaluated on three web projects at *LaBox*, a Spain-set web development company. Next, we describe each case in terms of three variables: the customer, the organization, and the web application under test (WAUT). Subjects were already playing the role of "customer" for the application at hand. Each unit of analysis was conducted at the customer's place. Table 3 profiles each subject along the main stumbling blocks identified in Section 3.

Unit of Analysis 1

- The customer (C1): she has no programming knowledge. She is used to diagramming tools and to test form-intensive Web applications. She works for a quite large company where she is usually asked to test new application interfaces. However, she does not like testing applications and complains about her work not being valued. She usually performs the testing on her own.

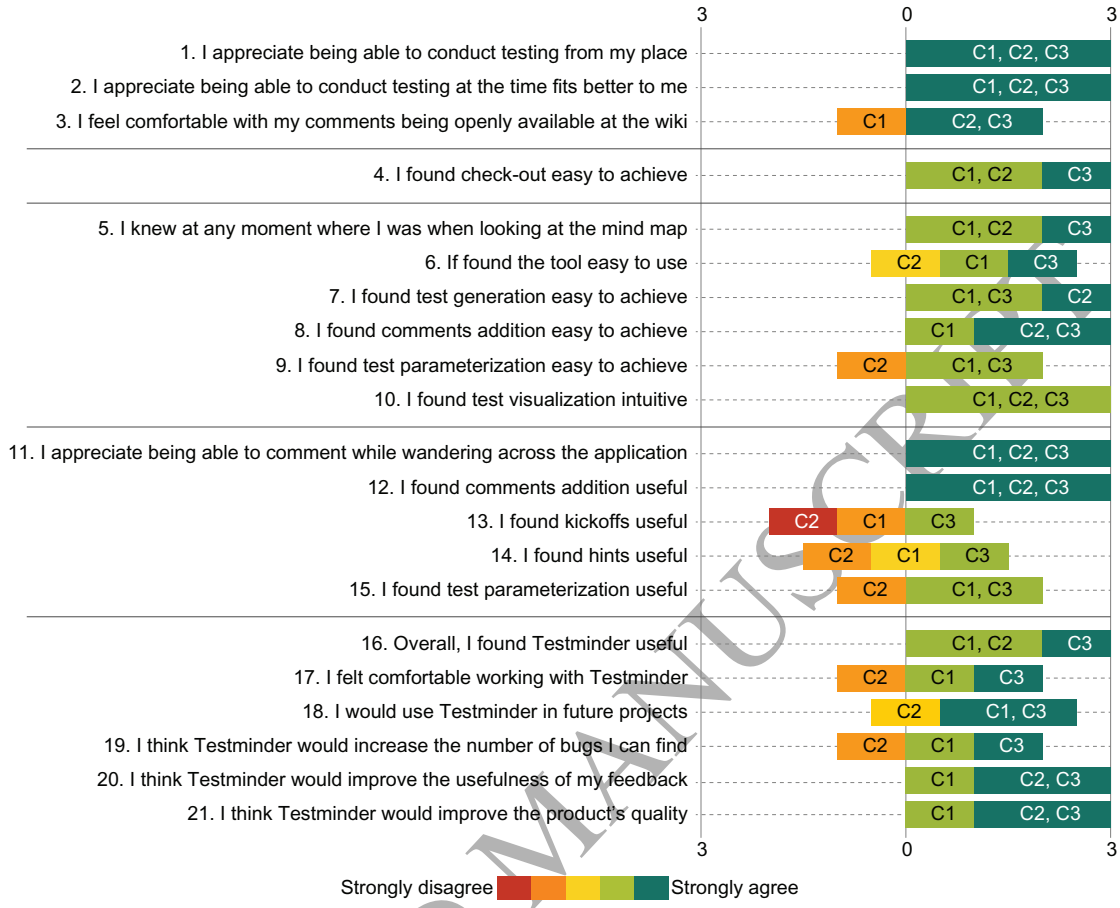


Figure 10: Diverging Stacked Bar Chart for the Satisfaction Questionnaire using Likert scales. The "3" on the left means the three customers, i.e. C1, C2 and C3, *Strongly Disagree* while "3" on the right corresponds to all *Strongly Agree*.

	Lack of Time	Lack of Motivation	Lack of Knowledge
C1	- no time assigned for testing - manual feedback generation is lengthy	- feels the effort of testing is not recognized - feels like a waste of time - feels she could be bothering developers	- no programming skills - low testing skills
C2	- very busy	- difficulty in describing problems to technical people	- no programming skills - no testing skills - phobia to technical concerns
C3	- manual UAT documentation is cumbersome	- very motivated	- low programming skills - low testing skills - used to mind mapping

Table 3: Subject profiling along risen issues.

- The organization: small-medium enterprise. The customer regretted her company not considering testing as a first-class duty as long as testing is not scheduled as part of her regular workload. As for testing practices, when an error is found she has to write an incidence describing the problem and send it through a web form to the development team. The customer confesses that the procedure was so tiresome that she occasionally let go some errors only not to create a new incidence. Her workplace and the testing place are 1,5 kilometers away.

- The WAUT: a human-resource management program. It took around 6 months to develop. At the onset, new versions were launched every month, but at the end of the project they had to test a new release every week. The application is for internal use. The potential number of users is expected to be around 100 people.

Unit of Analysis 2.

- The customer (C2): she has no programming skills. She finds it difficult to use new software, resorting to friends when facing "the deviltries" of her PC. However, she is the owner of the application. This makes her very conscious about the financial costs of bad testing. She is determined to make the application work since her business is going to go online. She claims to be always very busy.

- The organization: a Pharmacy. Testing feedback was based on making phone calls to developers when problems arose. The customer admitted having difficulties in explaining the problem, and that developers usually asked her to send screen captures to clarify the error. The Pharmacy is around 3 kilometers away from the testing place, i.e. the development company.

- The WAUT: an online store. It was developed in 2 months, having a new version available to test each week.

Unit of Analysis 3

- the customer (C3): she has no technical schooling, though she is familiarized with HTML and CSS. She uses *FreeMind* on a daily basis. She is very interested in technical concerns and she has extensive experience in testing Web projects. She finds testing very important, and she usually plays also the role of project manager. She admitted being very meticulous when checking, specially in visual details (e.g., element position, colors) and Web forms (e.g., wrong phone numbers, wrong email addresses). She spends a lot of time preparing accurate feedback documentation for developers.

ISSUES	Lack of Time	Lack of Motivation	Lack of Knowledge
MR1_async	✓	??	
MR2_R&R	??		✓
MR3_scaf			‰
MR4_maps			✓
MR5_mapping		‰	✓
MR6_feedback	✓	??	
MR7_learnability	??		??

Table 4: Validating meta-requirements as effective (✓), partially effective (‰) or unconclusive (??) as for the issue at hand.

- The organization: two freelance people. Their office is around 50 kilometers away from the testing place.

- The WAUT: a website offering online services to promote tourism. Her last project was developed in a hurry, lasting only 2 weeks. During these weeks she had to test new releases every day while the development team continued working against the clock.

Next subsections revise the causes of limited customer involvement in the light of the combined use of wikis and mind maps. Table 4 outlines the main constructs of the theory. For understanding sake, evaluation results are simultaneously presented.

8.3. RQ1. Does TestMind *alleviate the lack of time?*

8.3.1. Hypothesis

TestMind aims at easing the communication between customers and developers by asynchronously sharing the testing information via a wiki (i.e. *MR1_ async*). On the upside, asynchronicity relieves the team from the burden of trying to coincide in time and space. Furthermore, the customer might feel more free to wander around the application without being overseen by developers. On the downside, leaving the customer on his own might eventually lead to procrastination, more to the point if testing is felt to be an ancillary and not specially rewarding endeavour. In addition, R&R facilities (MR2_R&R) and automatic wiki page generation from *test maps* (MR6_feedback) might help to streamline the UAT process. Due to the short duration of the sessions and to the presence of the researcher, the learnability (MR7_learnability) was not measured on this first evaluation approach.

8.3.2. Result

Asynchronicity was highly valued. Though distance was not such a big issue, subjects valued the fact of "playing with the application on their own without the surveillance of developers". As a subject herself put it: "it never hurts to have a first go on my own, and if any doubt, I still can go back to direct contact. This, at least, saves me two or three trips". This seems to suggest *TestMind* to complement rather than substitute face-to-face meetings.

An interesting fact brought about by C2 was the fear of being "an annoying person". C2 is used to calling developers frequently. The fear of interrupting developers discourages her from notifying "some tiny issues that might not be worth the attention". Phone calls were regarded as an aggressive way of interacting, dissuading customers, and hence reducing the possibility of spotting improvement opportunities. C2 and C3 specially appreciated the use of *testing hints* to avoid important application spots to go unnoticed.

C1 and C3 found very useful UAT sessions being automatically exported as *FitNesse* pages. Traditionally, they spend much time trying to describe the encountered problems. C1 was particularly eager about this feature. The fact of testing tasks having poor recognition makes her unwilling to dedicate too much time to documenting. Indeed, C1 recognized that the effort for reporting mismatches was greatly reduced.

8.4. RQ2. Does TestMind *alleviate the lack of motivation?*

8.4.1. Hypothesis

TestMind facilitates testing to be conducted at the customer's workplace (MR1_async). This might well promote the visibility of testing, the spontaneous collaboration of the colleagues next door, and a better replication of the context in which the application will be used. By conducting testing at the customer's place, chances are that the customer's colleagues become aware of the testing effort. On the downside, this practice might be hindered by interruptions. Testing is not just running the application but also foreseeing different scenarios of usage. This requires focus to come up with different input data combinations or alternative navigation narratives. If the workplace carries a high likelihood of being interrupted, the benefits of in-place testing might be at jeopardy. In addition, reducing the burden of documentation (MR6_feedback) and rephrasing UAT as mind mapping (MR5_mapping) can make UAT a more enjoyable activity, hence fighting back the boredom that customers might feel with traditional UAT.

8.4.2. Result

This case study was not particularly appropriate to check for motivation. C2 and C3 were the owner and the project manager of their projects, respectively, so they were deeply committed. Only C1 expressed dissatisfaction for conducting test tasks right from the beginning. That said, *TestMind* can bring about "playability". C1 enjoyed "playing" with different data sets, and seeing the outcome in seconds. This was felt as more attractive that the boring task of running the application and next, reporting the experience by typing incidents into a Word document. This leads us to tick with a cautious ‰ the impact of *MR5_mapping* in Table 4. Questions 16 to 21 of the questionnaire (see Figure 10) attempt to get some insights on whether customer perception about testing itself changed as a result of using *TestMind*. When compared with respect to their previous face-to-face experiences, customers regard the *TestMind* experience as more effective (number of bugs caught, feedback usefulness, product quality impact). This could well motivate their engagement. So far, however, this has not been evaluated and hence, no claim is made about *TestMind* tackling the lack of motivation.

8.5. RQ3. Does TestMind *alleviate the lack of knowledge?*

8.5.1. Hypothesis

TestMind aims at empowering customers to conduct UAT on their own. Without their actions being supervised, customers might feel more relaxed and spend time exploring rarer data sets or different click streams. However, freedom also implies more involvement on the customer's side. This is a critical issue since the customer role tends to be transient, i.e. it is frequently played by different employees depending on the application being developed. Hence, the time dedicated to learn *TestMind* should pay off for a single application.

8.5.2. Result

Reducing the learning barrier was a main driver during *TestMind* development. Strategies include: (1) the use of mind maps as the notation for capturing testing sessions (MR4_maps); (2) resorting to a popular mind map editor, *FreeMind*, to increase the chances of customers being already familiarized with the interface; (3) sticking to *FreeMind* gestures to handle test maps in order to reduce the cognitive overload (MR5_mapping); (4) providing a headstart by obtaining test cases through R&R (MR2_R&R); and (5), the use of *test hints* and *Kickoffs* as a means to gently guide customers (MR3_scaf). Although results were promising, a more extensive evaluation would be required so as to ascertain the general learnability of the proposal (MR7_learnability).

None of the subjects presented big trouble understanding *test maps*. The radial disposition of nodes and the use of screenshots associated with page nodes was recognized as helpful for localization purposes (MR4_maps). Moreover, subjects appreciated the test recording utility (MR2_R&R). C1 and C3 edited the mind map for data set and comment addition. By contrast, C2 understood the parameterization and test reproducibility benefits, but she did not feel comfortable doing it. Rather, she would have liked what she called *"TestMind for dummies"* with the functionality being limited to wander, comment and export to *FitNesse*. She also believes on developers to check form validations better than herself.

The three of them enjoyed the comment-addition toolbar on the browser. They liked being able to add in-place suggestions during the recording without having to swap to another program.

Letting customers introduce their own input data sets might provide important cues to developers who can later

complement those sets with extra data. The notion of "expectation" proved to be an adequate way to informally capture testing assertions. Nevertheless, some subjects experimented problems in the two-stage definition of expectations: the *ExpectationPlaceholder* (at *Roaming* phase) and the *ExpectationStatement* (at *Commenting* phase). Some subjects were impelled to introduce the oracle (i.e. whether the current rendering matches expectations) at recording time, rather than waiting till replaying.

One shocking result was the poor valuation of *Kickoffs* (MR3_scaf). One customer strongly disagreed on *Kickoff* selection being a duty of the customer. Rather, she considered the application should be ready to test from the onset without forcing her to ascertain which *Kickoff* to choose. Alternatively, one subject expressed her willingness to participate in the design of the *Kickoffs*. So far, developers designed *Kickoffs* without any customer involvement.

8.6. Threats to validity

We follow here the recommendations of Runeson et al. as for analysing the extent to which the results are true and not biased by the researchers' subjective point of view [47].

Construct validity refers to the appropriateness of inferences made on the basis of observations or measurements (often test scores), specifically whether a test measures the intended construct [47]. In this work, this mapping goes as follows:

1. questions 1 - 3 provide feedback as for asynchronicity (MR1_async)
2. question 4 collects information as for automating documentation generation (MR6_feedback)
3. questions 5 - 10 provide evidences on lowering the UAT effort (MR2_R&R, MR4_maps, MR5_mapping)
4. questions 11 - 15 supply insights for in-place assistance (MR3_scaf)

Internal validity is a matter of concern when causal relationships are examined. It depends upon whether the observed change in a dependent variable is, indeed, caused by a corresponding change in an independent variable and not by other factors. Here, the ability to conduct UAT at the customer's location and at their most suitable time, both have a high internal validity with regard to lessen the lack of time. So does lowering the learning bar and UAT assistance with regard to lessen the lack of knowledge.

However, it should be noted that experiments were conducted in a single session, scheduled in advance. Customers were not free to allocate the UAT at their most convenient time. Thus, procrastination risks were not assessed. Subjects' timely response was, probably, facilitated by the excitement of the novelty together with the presence of the researcher. In a real situation both boosters will not be there. Nevertheless, leaving customers largely on their own with the only pressure of a deadline might be risky, more to the point if UAT is regarded as an ancillary activity. Fortunately, developers might track customer testing progress through the wiki. If there is a delay, developers can resort to email or phone calls to remind customers of their testing duty.

Another issue for self-paced UAT is focus loss. UAT sessions should focus on the Sprint features to be tested. However, our experience is that customers are very often too tempted to wander around the application, leading to large UAT sessions that hinders developers from grasping the real intention of the customer. *TestMind* does not prevent wandering around but might warn about it. A proposed improvement is to count the number of *Action* nodes in the *UATCase* and, if a certain threshold is exceeded, show a warning message. Both issues, i.e. procrastination and focus loss, are left for future evaluations.

Evaluation was performed by only one researcher. To minimize the possible bias, customer sensations were gathered through two different means: the researcher's notes and a Likert questionnaire. The researcher's notes were checked against the scale ratings obtained in the questionnaire so as to verify the customer impressions.

External validity is concerned with the extent to which the problem and the findings can be generalized. A common criticism of case studies is that their results only apply to the instance being studied. We tried to reduce this threat by selecting diverse customer profiles (different technical background, occupation, web application sizes). Nevertheless, the three evaluated cases involve a single customer. We have not evaluated the case where the same feature is tested by several customers. However, we do not envision much difference in the results unless customers gather together to conduct the test. Nevertheless, a larger group of customers is still required to sustain the results of this work.

Reliability is concerned with the extent to which the data and the analysis are dependent on the specific researchers. Hypothetically, if another researcher conducted the same study at another time, the result should be the same. We reduce this threat by describing the subjects of the studies in terms of customers, organizations and WAUT application, as well as presenting the questionnaire. In this way, other researchers can recreate this experiment in the same setting and check the results.

9. Architecture

Figure 11 outlines the *TestMind* architecture. Broadly, *TestMind* follows a model-view-controller architecture:

- the *TestMap* model (4). This model captures the UAT session data along the meta-model depicted in Figure A.13.
- the view (1). Test maps are visually rendered as mind maps in *Freemind*.
- the controller. It achieves a double aim. First, it keeps the *FreeMind canvas* in sync with the underlying *TestMap* model. This is achieved through the

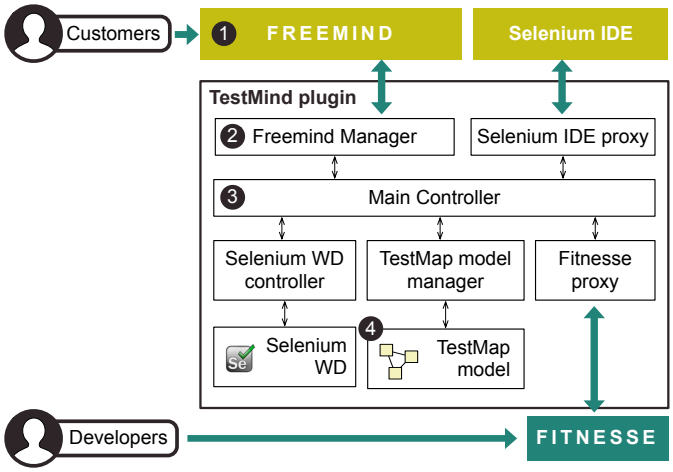


Figure 11: *TestMind* architecture

FreeMind manager (2), which hooks the *TestMind* plug-in to *FreeMind*. In addition, it listens to child creation interactions (e.g., TAB key pressing) and customizes these interactions to account for the test map semantics. This customization might trigger interactions with *FitNesse* or *Selenium* that happen through the Main Controller (3).

Installation guidelines for *TestMind* can be found in Appendix B.

10. Related Work

Different approaches and techniques have been devised to foster customer participation in testing. This section reviews some of these works w.r.t. *TestMind*.

Domain Specific Languages (DSLs). Abstraction is a common mechanism to reduce the gap between technical and non-technical people. The use of DSLs to define test cases improves their readability, making them understandable to different stakeholders, from developers to business experts. Toolkits such as *Cucumber* [12] or *FIT/Fitnesse* [33, 11] resort to DSLs for describing *test stories* which can later be converted into executable test cases [37]. Within *FIT*, the *Telling TestStories* tool integrates test generation techniques (such as model-based testing) for the test and test data specification to be described in a tabular form [48]. Häser et al. [49] propose the integration of business domain concepts into testing DSLs. They conclude that including these concepts permit a faster creation of test case specification. TestMind aligns with this research as for the use of DSLs, specifically, the use of mind maps (the DSL's graphical concrete syntax) for UAT.

Live demos. They allow developers to show the application to customers, collecting their opinions and suggestions [24]. This approach might fail to cover all scenarios since demos themselves tend to be led by developers, hence representing their mindset on what the application is about, rather than the customers'. Hence, live demos are recommended to be jointly used with real releases where customers are left on their own "playing with the application", i.e. manual testing [24].

Manual testing. This is a widely accepted practice for UAT [3]. Here, users wander through the application coming with examples and action sequences not necessarily foreseen by developers. Traditionally, feedback is given via emails or phone calls. In this line, *JIRA Capture* [50] is a tool that permits users to annotate application screenshots while navigating. Then, JIRA incidences are automatically created from them. In most cases, the quality of the feedback greatly depends on customer's knowledge and motivation. In some cases, it may be difficult for developers to reproduce the UAT scenario in order to clarify the root of the problem [3]. This sustains the vision of UAT as a collaborative endeavour between developers and consumers, and underlines the importance of the testing scaffold. This is aligned with Shaye's vision of customers developing and executing automated tests by themselves without resorting to the technical staff [17]. *TestMind* shares this mindset.

Record&Replay tools. They target non-technical customers to generate tests out of browsing sessions. Unfortunately, the lack of parameterization and modularity mechanisms make so-generated test scripts difficult to maintain. In these scripts, input data is hardcoded while the test structure is strongly coupled with the web interface. The latter makes these scripts fragile upon changes to the application GUI [51]. *TestMind* departs from R&R tools by providing a context. In *TestMind*, Selenium IDE [36] is not launched in a vacuum but as part of a larger UAT session that aims at obtaining a *test map*. A test map can include different Selenium scripts that are complemented with input data sets, screenshots, comments and expectations, all arranged along a mind map structure, hence facilitating location and experimentation.

Test automation. It is "the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes" [52]. *TestMind* is a software that helps the capture of Selenium scripts and their enactment, and from this perspective, it can be framed within the test automation effort. However, the aim of *TestMind* is not so much about automation but about asynchronous collaboration. Tests need to be enactable not so much from being later automated but to be replayed by developers and hence, to get a better insight of customers' UAT session. That said, *test maps* could be used as a sort of regression tests for next Sprints. In a survey conducted in [53], regression testing was the most frequently mentioned factor for test automation decisions. This makes sense in an Agile setting where previously developed backlog features need to be verified to check whether they still perform correctly on the next Sprint's release. Unfortunately, small changes on the application UI can make recorded test scripts to fall apart. This backward-compatibility requirement in GUI

evolution might be too stringent in some scenarios, and might increase the maintenance cost of keeping *test maps* up and running.

11. Conclusions

Agile methodologies put stringent demands on UAT, if only for the frequency at which it needs to be conducted. In-person meetings might need to be complemented with asynchronous ways for customers and developers to collaborate during UAT. We coin the term "self-paced UAT" to denote asynchronous sessions where customers perform UAT on their own using a scaffolding previously set by developers. Test scaffolding helps customers to effectively perform UAT (keeping the focus through *testing hints*) and efficiently (automatically setting customers in ready-to-go scenarios through *kickoffs*). In addition, mind maps are proposed to give structure and context to UAT sessions. In this way, Record&Replay is not launched in a vacuum but framed within a *test map*.

First evaluations are promising. Subjects specially valued the chance of conducting UAT at their own pace. No travel, no agenda sync problems. They all prized the opportunity to add comments during test recording (the best rated feature in Likert scales) and to report feedback with a single click. Test parameterization was specially appreciated by the subjects who usually checked form-intensive websites.

Future work will look at the extend to which *test maps* can be reused as regression tests on future Sprints. Here, we envisage the advent of backward-compatibility issues. Also, a more extensive literature review would help to better support the presented list of issues on customer involvement. Finally, additional evaluation is needed. Specifically, we would like to look how procrastination can jeopardize the agility of UAT. The impact of self-paced UAT on customers' motivation is also worth investigating.

Acknowledgements

We are indebted to the anonymous reviewers for providing insightful comments and pointing out most recent literature. We would like to thank Philipp Brune for his encouragement throughout, and Antoni Olivé for introducing us to Design Science. This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2014-58131-R.

References

[1] Manifesto for Agile Software Development, `http://www.agilemanifesto.org`, [29 February 2016].

[2] E. F. Collins, V. F. de Lucena, Software Test Automation practices in agile development environment: An industry experience report, in: Automation of Software Test (AST), 2012 7th International Workshop on, 2012, pp. 57–63. doi:10.1109/IWAST.2012.6228991.

[3] G. Liebel, E. Alegroth, R. Feldt, State-of-Practice in GUI-based System and Acceptance Testing: An Industrial Multiple-Case Study, in: Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on, 2013, pp. 17–24. doi:10.1109/SEAA.2013.29.

[4] R. Hoda, J. Noble, S. Marshall, The impact of inadequate customer collaboration on self-organizing Agile teams, Information and Software Technology 53 (5) (2011) 521–534. doi:10.1016/j.infsof.2010.10.009.

[5] B. Haugset, G. K. Hanssen, Automated Acceptance Testing: A Literature Review and an Industrial Case Study, in: Agile, 2008. AGILE '08. Conference, 2008, pp. 27–38. doi:10.1109/Agile.2008.82.

[6] T. Issa, P. Isaias, User Participation in the System Development Process, Springer London, London, 2015. doi:10.1007/978-1-4471-6753-2.

[7] B. Meyer, Agile!, Springer International Publishing, Cham, 2014. doi:10.1007/978-3-319-05155-0.

[8] J. Kupersmith, Putting the User Back in User Acceptance Testing, `http://www.modernanalyst.com/Resources/Articles/tabid/115/articleType/ArticleView/articleId/617/Putting-the-User-Back-in-User-Acceptance-Testing.aspx`, [29 February 2016].

[9] J. Itkonen, M. V. Mäntylä, C. Lassenius, The Role of the Tester's Knowledge in Exploratory Software Testing, IEEE Transactions on Software Engineering 39 (5) (2013) 707–724. doi:10.1109/TSE.2012.55.

[10] I. Lykourentzou, F. Dagka, K. Papadaki, G. Lepouras, C. Vassilakis, Wikis in enterprise settings: a survey, Enterprise Information Systems 6 (1) (2012) 1–53. doi:10.1080/17517575.2011.580008.

[11] FitNesse, `http://www.fitnesse.org`, [29 February 2016].

[12] Cucumber, `https://cukes.info`, [29 February 2016].

[13] P. Johannesson, E. Perjons, An introduction to design science, Springer, 2014.

[14] Investopedia, Acceptance Testing, `http://www.investopedia.com/terms/a/acceptance-testing.asp`, [29 February 2016].

[15] A. Alliance, Acceptance Testing, `http://guide.agilealliance.org/guide/acceptance.html`, [29 February 2016].

[16] S. S., What is User Acceptance Testing (UAT) and How to Perform It Effectively?, `http://www.softwaretestinghelp.com/what-is-user-acceptance-testing-uat/`, [26 October 2016] (2016).

[17] S. D. Shaye, Transitioning a Team to Agile Test Methods., in: AGILE, 2008, pp. 470–477.

[18] R. Rasmussen, A. S. Christensen, T. Fjeldsted, M. Hertzum, Selecting users for participation in {IT} projects: Trading a representative sample for advocates and champions?, Interacting with Computers 23 (2) (2011) 176–187. doi:10.1016/j.intcom.2011.02.006.

[19] K. Beck, Extreme programming explained: embrace change, Addison-Wesley Professional, 2000.

[20] A. Martin, R. Biddle, J. Noble, Agile Software Development: Current Research and Future Directions, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, Ch. An Ideal C, pp. 111–141. doi:10.1007/978-3-642-12575-1_6.

[21] B. Linders, Requirements with the Agile process management, `http://searchsoftwarequality.techtarget.com/tip/Requirements-with-the-Agile-process-management`, [13 October 2016] (2016).

[22] G. K. Hanssen, B. Haugset, Automated Acceptance Testing Using Fit, in: System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on, 2009, pp. 1–8. doi:10.1109/HICSS.2009.83.

[23] M. Puleio, How not to do agile testing, in: Agile Conference, 2006, 2006, pp. 7 pp.–314. doi:10.1109/AGILE.2006.34.

[24] J. Shore, S. Warden, The art of agile development, O'Reilly Media, Inc., 2007.

[25] R. W. Rice, What is User Acceptance Testing?, `http://www.riceconsulting.com/articles/what-is-UAT.htm`, [29 Febru-

ary 2016].
[26] K. N. Johnson, Tips for better user acceptance testing, http://www.informit.com/articles/article.aspx?p=1431821&seqNum=5, [29 February 2016] (2009).
[27] L. Cao, B. Ramesh, Agile Requirements Engineering Practices: An Empirical Study, IEEE Software 25 (1) (2008) 60–67. doi:10.1109/MS.2008.1.
[28] P. Louridas, Using wikis in software development, IEEE Software 23 (2) (2006) 88–91. doi:10.1109/MS.2006.62.
[29] D. Ferreira, A. R. da Silva, Wiki supported collaborative requirements engineering, in: Wikis4SE 2008 Workshop, Porto, Protugal, 2008.
[30] I. Hadar, M. Levy, Y. Ben-Chaim, E. Farchi, Using Wiki as a Collaboration Platform for Software Requirements and Design, Springer International Publishing, Cham, 2016, pp. 529–536. doi:10.1007/978-3-319-27478-2_40.
[31] J. Rech, C. Bogner, V. Haas, Using Wikis to Tackle Reuse in Software Projects, IEEE Software 24 (6) (2007) 99–104. doi:10.1109/MS.2007.183.
[32] C. McMahon, Wikis for Agile ALM Collaboration, http://searchsoftwarequality.techtarget.com/tip/Wikis-for-Agile-ALM-collaboration, [29 February 2016] (2008).
[33] R. Mugridge, W. Cunningham, Fit for developing software: framework for integrated tests, Pearson Education, 2005.
[34] L. Crispin, FitNesse: A Tester's Perspective, http://www.methodsandtools.com/tools/tools.php?fitnesse, [13 October 2016].
[35] B. Haugset, G. K. Hanssen, The Home Ground of Automated Acceptance Testing: Mature Use of FitNesse, in: Agile Conference (AGILE), 2011, 2011, pp. 97–106. doi:10.1109/AGILE.2011.37.
[36] Selenium IDE, http://www.seleniumhq.org/projects/ide/, [29 February 2016].
[37] F. Ricca, M. D. Penta, M. Torchiano, P. Tonella, M. Ceccato, C. A. Visaggio, Are fit tables really talking?, in: 2008 ACM/IEEE 30th International Conference on Software Engineering, 2008, pp. 361–370. doi:10.1145/1368088.1368138.
[38] Mind map, https://en.wikipedia.org/wiki/Mind_map, [29 February 2016].
[39] T. Buzan, Mind maps for business : using the ultimate thinking tool to revolutionise how you work, Pearson, Harlow, 2014.
[40] M. Mernik, J. Heering, A. M. Sloane, When and How to Develop Domain-specific Languages, ACM Comput. Surv. 37 (4) (2005) 316–344. doi:10.1145/1118890.1118892.
[41] IEEE, Systems and software engineering – Vocabulary, ISO/IEC/IEEE 24765:2010(E) (2010) 1–418doi:10.1109/IEEESTD.2010.5733835.
[42] L. Baresi, M. Young, Test oracles, Techn. Report CISTR-01 2 (2001) 9.
[43] C. Kaner, The ongoing revolution in software testing, in: Software Test & Performance Conference, Vol. 8, 2004.
[44] Freemind, http://freemind.sourceforge.net/wiki/index.php/Main_Page, [29 February 2016].
[45] W3C, HTML and XHTML Techniques for WCAG 2.0, https://www.w3.org/TR/WCAG20-TECHS/html.html, [29 February 2016].
[46] P. Runeson, M. Host, A. Rainer, B. Regnell, Case study research in software engineering: Guidelines and examples, John Wiley & Sons, 2012.
[47] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2) (2009) 131–164. doi:10.1007/s10664-008-9102-8.
[48] M. Felderer, P. Zech, F. Fiedler, R. Breu, A Tool-Based Methodology for System Testing of Service-Oriented Systems, in: Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on, 2010, pp. 108–113. doi:10.1109/VALID.2010.12.
[49] F. Häser, M. Felderer, R. Breu, Is business domain language support beneficial for creating test case specifications: A controlled experiment, Information and Software Technology 79 (2016) 52–62. doi:10.1016/j.infsof.2016.07.001.
[50] Capture for JIRA, https://es.atlassian.com/software/jira/capture, [29 February 2016].
[51] M. Leotta, D. Clerissi, F. Ricca, P. Tonella, Capture-replay vs. programmable web testing: An empirical assessment during test case evolution, in: Reverse Engineering (WCRE), 2013 20th Working Conference on, IEEE, 2013, pp. 272–281. doi:10.1109/WCRE.2013.6671302.
[52] D. Huizinga, A. Kolawa, Automated defect prevention: best practices in software management, Wiley-IEEE Computer Society Press, 2007.
[53] V. Garousi, M. V. Mäntylä, When and what to automate in software testing? A multi-vocal literature review, Information and Software Technology 76 (2016) 92–117. doi:10.1016/j.infsof.2016.04.015.

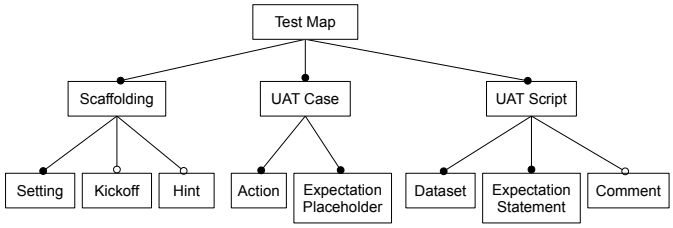


Figure A.12: UAT Feature Model.

Appendix A. Designing Test Maps

Test maps are mind maps but not all mind maps are test maps. That is, test maps restrict the structure and kind of participating nodes. In short, test maps become the graphical representation of a Domain Specific Language (DSL) for UAT. Coming up with a DSL implies identifying the main concerns of the UAT domain (i.e. the feature model). Next, providing a metamodel that captures the main relationships (a.k.a. abstract syntax), and finally, facilitating a graphical notation to specify the DSL expressions (a.k.a. concrete syntax). We begin with the feature model.

The feature model. In the analysis phase of DSL development, the problem domain is identified and domain knowledge is gathered [40]. Broadly, the output consists basically of domain-specific terminology and semantics in more or less abstract form, being feature models a the main asset. A feature model captures "the commonalities and variabilities of domain concepts and their interdependencies" [40]. In this case, "the domain" is a UAT session. Figure A.12 depicts the main UAT concerns for our purposes. This diagram states that a *test map* captures a UAT session in terms of the available **scaffolding** set for guiding the session together with a **test case** and different enactments of this test, i.e. **test scripts**. These features where already described in Section 6. Notice that the aim is to identify the main concerns and their variabilities. How these concerns are to be expressed is postponed till the abstract syntax is specified.

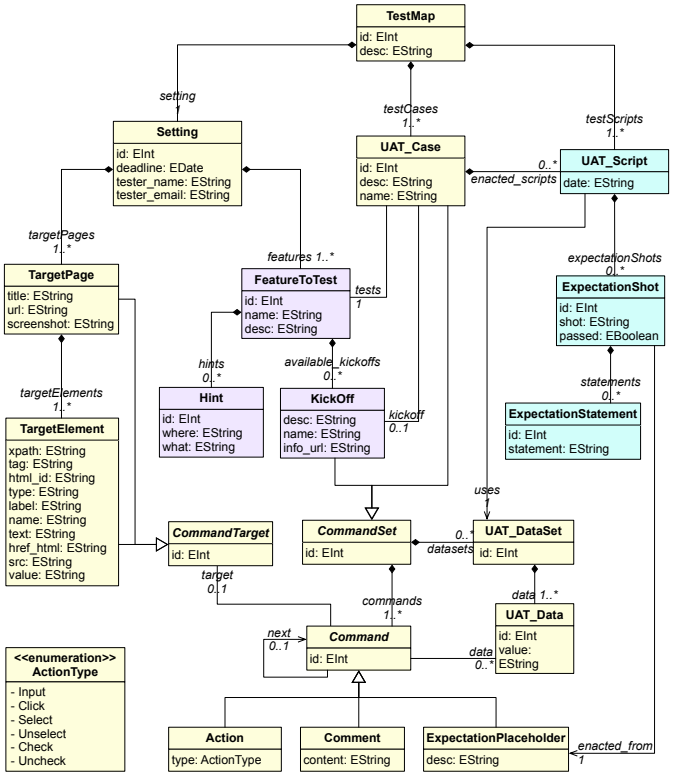


Figure A.13: TestMind Abstract Syntax.

Abstract Syntax. Concerns risen during DSL analysis should now find their way into the DSL's abstract syntax. The abstract syntax describes the concepts of the language, the relationships among them, and the structuring rules that constrain the model elements and their combinations in order to respect the domain rules. This is expressed as the DSL metamodel (see Figure A.13). A TestMap model (i.e. a *test map*) includes five main classes, namely:

- the *Setting* class, which indicates the name and email of the customer performing the testing, and a deadline for conducting the UAT. This class also contains the features to be tested by this specific customer.
- the *Kickoff* class, which holds the different Kickoff scenarios for each feature.
- the *Hint* class, which describes tooltip notes to be displayed during test case recording for a specific feature.
- the *UAT_Case* class, which stands for a test case. Test cases are meant to be provided through R&R tools. Hence, this class's expressiveness is based on the R&R tool to be used: Selenium IDE. Test cases are defined as a set of commands that mimic the interactions the customer performed during the test recording. Every command might interact with a target element (*CommandTarget*), that can be either a web element or the whole web page. Commands might be of three types: *Actions*, *Comments* and *ExpectationPlaceHolders*.
- the *UAT_Dataset* class, which collects a set of input data (*UAT_Data*) to be used on each test case execution in order to check the test case on different scenarios.
- the *UAT_Script* class, which stores the result of the execution of a *UAT_Case* using the provided *UAT_Datasets*. This class includes the *ExpectationShots* taken for the defined *ExpectationPlaceholders* on each execution.

Concrete syntax. The concrete syntax comprises a mapping between the metamodel concepts (i.e., the abstract syntax) and their textual or visual representation. While the abstract syntax addresses expressiveness, the concrete syntax cares for usability as for the target audience. Our target audience are customers with low or no technical knowledge. We then resort to mind maps as a notation of test maps. However, though *test maps* are mind maps, not all mind maps are *test maps*. That is, we need to restrict the expressiveness of mind maps to limit the kind of nodes and the structure that *test maps* can exhibit.

Limiting the kind of nodes. Mind mapping supports the notion of *Node*. This general notion needs to be specialized into the different concerns risen during UAT. Hence, we do no longer have just generic nodes but *Kickoff* nodes, *Action* nodes, *UATCase* nodes and so on. The type of each node is denoted through an icon (see Figure 5). Icons play the same role than profiles in UML. In this way, classes in the abstract syntax are mapped as nodes with specific icons. Concepts are mapped to nodes where the node's icon stands for the type and the label and associated notes accommodate the rest of the properties.

Limiting the structure. The arrangement of the previously defined node types should follow some rules to maintain the *UATCase* structure. For example, *Action* nodes can hang from *Pages* but not vice versa, *UATScript* nodes can hold *ExpectationShot* nodes but not *Kickoff* nodes, etc [6].

Appendix B. TestMind installation guide

TestMind installation goes along three steps:

1. install *FreeMind*[7]
2. install *TestMind* for Freemind[8] and configure it for your *FitNesse* installation

[6]This is supported by extending the FreeMind's XML Schema with additional sub-types and restrictions along the TestMind's abstract syntax

[7]http://freemind.sourceforge.net/wiki/index.php/Download accessed 29-Feb-16.

[8]http://www.onekin.org/testmind accessed 15-Nov-16.

3. install *Selenium IDE*[9] and its *TestMind* plug-in[10] to enhance *Selenium IDE* with the *hint bar*, i.e. a bar where testing hints are showed; and the *commenting bar*, i.e., a bar where user comments can be added during navigation.

TestMind has been checked with *FreeMind* v1.0.1, *Firefox 37, Selenium IDE* v2.9.1, and *FitNesse* v20130530.

Appendix C. Interview Guide

1. Was the subject alone when conducting the test? Were other colleagues around?
2. How quiet was the place when conducting the test? (phone calls, WhatsApp sounds, etc.)
3. How many interruptions happened when conducting the test? (phone ringing, door slams, chatting staffers, etc)
4. What technical resources were there? (Screen size, wifi, broad band, ...)
5. Was the subject capable of creating valid tests?
6. Did he introduce any error into the test?
7. How many tests did he generate?
8. How many comments did he add during the recording?
9. How many ExpectationPlaceholders did he use?
10. Where did he find most problems?
11. What benefits would he highlight from TestMind?
12. Is the subject easily finding the features?
13. Is the subject feeling comfortable using the tool?
14. Does the subject feel eager to create tests with this tool?
15. Does the subject understand the mind map view of the tests?
16. Will the subject prefer a different visualization?
17. Will the subject change anything on the tool?

Vitae

Itziar Otaduy is a PhD student at the University of the Basque Country (UPV/EHU). Otaduy obtained a BSc in Computer Engineering and a MSc in Advanced Computer Systems at the University of the Basque Country. Her current research interests include Web Testing and Test Automation. Contact her at itziar.otaduy@ehu.eus.

Oscar Díaz is Full Professor at the University of the Basque Country (UPV/EHU). He leads the ONEKIN group, with a focus on Web Technologies and close partnership with industry. He obtained the BSc in Computing at the University of the Basque Country, and a PhD by the University of Aberdeen. His current interests include Software Product Lines, Continuous Integration and Design Science. Contact him at Facultad de Informática, Apdo. 649, 20.011 San Sebastián (Spain), oscar.diaz@ehu.eus.

[9] http://www.seleniumhq.org/projects/ide accessed 29-Feb-16.
[10] http://www.onekin.org/testmind accessed 15-Nov-16.