Usability Improvement through A/B Testing and Refactoring

Sergio Firmenich[1], Alejandra Garrido[1,*], Julián Grigera[2], José Matías Rivero, Gustavo Rossi[1]

LIFIA, Fac. de Informática, Universidad Nacional de La Plata, Argentina
Calle 50 y 120 s/n, 1º piso, La Plata (1900), Pcia. de Bs.As., Argentina
[1]Also CONICET, Argentina
[2]Also CIC, Argentina
[*]Corresponding author ORCID: 0000-0002-5052-705X
Email: garrido@lifia.info.unlp.edu.ar Phone: +54 9 221 4228252

ABSTRACT

Usability evaluation is an essential task in web application development. There have been several attempts to integrate user-centered design with agile methods, but it is hard to synchronize their practices. User testing is very valuable to learn from feedback of actual use, but it remains expensive to find and solve usability problems. Furthermore, the high cost of usability evaluation forces small/medium-sized companies to trust the first solution applied, without actually testing the success of the solution or considering a possible regression in usability, as could be highlighted by an iterative testing method.

In this article we advocate for a usability improvement cycle oriented by user feedback, and compatible with an agile development process. We propose an iterative method supported by a toolkit that allows usability experts to design user tests, run them remotely, analyze results, and assess alternative solutions to usability problems similarly to A/B testing. Each solution is created by applying client-side web refactorings, i.e., changes to the web pages in the client which are meant to improve usability. The main benefit of our approach is that it reduces the overall cost of user testing and particularly, A/B testing, by applying refactorings to create alternative solutions without modifying the application's server code. By making it affordable for usability experts to apply the method in parallel with the development cycle, we aim to encourage them to incorporate user feedback and try different ideas to discover the best performing solution in terms of the metrics of interest.

Keywords: usability evaluation; A/B testing; web refactoring; agile methods; external quality.

Acknowledgments: The authors acknowledge the support from the Argentinian National Agency for Scientific and Technical Promotion (ANPCyT), grant number PICT-2015-3000.

1. INTRODUCTION

In many of our daily life activities the web is becoming pervasive, and yet, its acceptability and success can only come from the hand of usability. Today's research positions usability as one of the most important quality factors for web applications (Fernandez et al. 2011). Contradictorily, usability evaluation is often neglected, mainly because of its cost regarding both time and resources (Nielsen and Loranger 2006). On the one hand, there have been several attempts to integrate user-centered design (UCD) in agile development methods (Jurca et al. 2014; Salvador et al. 2014). These studies have found that UCD experts mostly use inspection methods manually, usually working on several projects at a time and in many stages of each project, and therefore cannot keep up with their work in time to keep ahead of the next release. On the other hand, when the focus is to learn from feedback of real end-users, empirical methods, especially user testing, is preferred over inspection methods (Rubin

and Chisnell 2008). In user testing, representative users evaluate a website generally by completing a sequence of typical tasks, which is costly regarding resources. To reduce costs, different tools have been developed to automate user testing in some way (Fernandez et al. 2011). The results that tools provide, depending on the strategy used, vary from measures of time that each user took to complete a task, traces of mouse movement, deviations from an optimal way of performing a task, satisfaction survey results, etc. A usability expert must analyze these results, discover usability problems (sometimes with a little help from tools) and manually apply the necessary changes to solve the problems.

Besides the lack of tools to apply the solutions, the usual process of user testing finishes when the developer applies the changes, which assumes that these changes actually solve the problems, and there's no need to prove it with a new cycle of user testing. These assumptions may well be false, and mainly they distant from current practice in agile development and maintenance cycles, where improvements are applied incrementally and testing is performed frequently. In contrast, A/B testing has built on the premise that frequent experimentation is indispensable since most of the ideas fail to improve key metrics (Kohavi and Longbotham 2015). Actually, studies show that only 10% to 30% of the ideas are successful, so failing fast allows the necessary course adjustments for more successful ideas to be proposed (Kohavi et al. 2009). Thus, A/B testing proposes to split the universe of visitors in two or more groups with the purpose of testing different versions of an idea with each group and comparing the effect of each version. While A/B testing is generally used to improve revenue, a similar approach may be used to improve usability. However, A/B testing is very expensive, since it requires constructing the different versions to test, defining the metrics of interest and how to compute them from user events, recording test results, and analyzing the results to find the best solution. Thus, trying to apply A/B testing for usability evaluation manually in the context of an agile development process would be close to unfeasible.

Upon the challenges previously described, our goals are:

- to integrate the idea of A/B testing in the context of an iterative and incremental method of usability improvement;
- to provide tools to help usability experts at every step of the usability improvement method, so they can apply the method in parallel and independently of the development cycle;
- to make the method feasible and compatible with an agile development process.

We have devised an *automated A/B usability testing and improvement method* that starts once there is a potentially shippable product increment. If this increment has been released, it may be exposed to a statistically significant number of users for a higher confidence in test results. Nevertheless, there are large benefits in testing as early as possible, to find the main usability problems before release. Nielsen suggests that running small tests early and often in the context of an iterative design is preferable over a single, elaborate and costly test (Nielsen 2000). Thus, we support user testing even with a small number of users as soon as there is a user interface available, i.e., a Minimum Testable Product (MTP). Moreover, integrating A/B testing allows that, when some usability problems have been exposed in the current version, different possible solutions can be tested and the best one can be selected on a solid ground, since it was experimentally proven to be the best. To minimize the cost of A/B testing and make it possible for the usability expert to craft different ideas in new versions of the

application, (and alleviate developers of this work) we propose to apply rapid, temporary solutions by means of *client-side web refactorings (CSWRs)* (Garrido et al. 2013b). We have defined CSWRs as modular and implemented solutions to usability problems, which apply transformations automatically to the Document Object Model (DOM) of web pages when installed on the client browser. The advantages of using CSWRs in the context of our usability improvement method are: *automation*, *rapid feedback* and *tracking*. Firstly, CSWRs may be easily instantiated from generic scripts by providing appropriate parameters, and installed in the browser to be automatically applied when the target page is loaded. Secondly, they are readily available to test a solution and to obtain immediate feedback of its real benefits. Thirdly, each CSWR is a first-class object in our model and it is associated with the task that is intended to improve. This provides precise tracking of the changes applied to each new version of the application and each user task.

Specifically, our A/B usability testing and improvement method has five stages, with the idea that the first four stages are covered by the usability expert (we'll call him UX expert or just expert for short), and only the fifth stage involves developers. In the **first** stage, the expert designs the user test, i.e., the task to be exercised, the test scenario and the metrics to be calculated during each test execution (which we call *observations*). The tools we have constructed to help the UX expert in this step are the *Scenario Recorder* and *Scenario Editor*. The test scenario and observations are fed into a tool called *Scenario Player*, which allows test subjects executing the test remotely, guiding them through the steps in the scenario while it records the results of the observations. In the **second** stage, the expert analyzes test results to identify usability problems, which may be accomplished using a *Test Analysis Tool* that shows the results in different charts and diagrams. Once the problems were identified, the expert may apply different combinations of CSWRs as solutions to the problems. For this, the *CSWR Framework* provides implemented refactorings and allows for their automated application in the client. The **third** stage of the method consists of user testing each new version of the task dividing the subjects in as many groups as versions, in a way similar to A/B testing or split testing. In the **fourth** stage the UX expert compares the test results of each version against each other and with the results of the first stage to determine the best solution. Finally, in the **fifth** stage developers receive the specification of the best solution (the winning combination of CSWRs) and implement this solution in the back-end, i.e., the best combination of CSWRs is coded in the main application. In the next iteration, the main application may be tested again for usability with another use case or group of tasks.

The rest of the paper is organized as follows. The next section provides background on user testing, usability refactoring and A/B testing, and discusses related work. Section 3 describes our method for the improvement of the usability of web applications and defines the models involved in the method. Section 4 presents the tools that we have built to automate or semi-automate each stage of the method. These tools are aligned with the models providing the traceability and automation necessary for the feasibility of the method. Section 5 shows our proposal to make the method applicable within an agile development process. Section 6 presents three case studies that evaluate different aspects of our approach and finally Section 7 presents conclusions and future work.

2. BACKGROUND AND RELATED WORK

This section firstly provides a background on usability evaluation methods, specifically about current approaches for user testing, followed by a description of web refactorings applied to improve usability, and A/B testing. Secondly, we present related work on iterative approaches to usability improvement, methods to integrate usability testing or UCD in agile development methodologies and other approaches that apply A/B testing for usability.

2.1 Background

A wide variety of ***usability evaluation methods (UEMs)*** have been proposed over the years, and some authors have proposed different ways to classify them and compare them. A broad classification divides UEMs in two main types: formative and summative (Hartson et al. 2003). *Formative evaluation* is performed by usability experts at an early design stage, to find usability problems on pre-release prototypes. Meanwhile, *summative evaluation* takes place after development, to assess or compare the level of usability achieved with the final design (Hartson et al. 2003). Fernandez et al. call the former *inspection methods*, and in their systematic mapping study, they propose to sub-classify them into heuristic evaluation, guideline review, cognitive walkthrough and perspective-based inspection (Fernandez et al. 2011). While formative evaluation using inspection methods is important in Model Driven Development, as remarked by Fernandez et al. in their own evaluation process (WUEP) (Fernandez et al. 2013), the kind of problems that these UEMs can find are limited to a lab setting, the quality of the guidelines and the expertise of the evaluator (Fernandez et al. 2011). Most importantly, they cannot learn from feedback of real end-users, as we advocate from the point of view of current practices in agile methods (Williams and Cockburn 2003).

The limitations of inspection methods have led to the popularity of empirical methods, particularly ***user testing***, which capture and analyze real usage data (Rubin and Chisnell 2008). In user testing, a usability expert observes and interviews a representative sample of end users while they perform typical tasks with the system, collects quantitative and qualitative measures and recommends improvements (Rubin and Chisnell 2008). The down-side of user testing is that it is expensive. On the one hand, it requires recruiting users and, on the other hand, experts must still analyze the results, transform them into usability problem descriptions, and find solutions for those problems that developers must implement by hand.

Back in 1998, Hartson and Castillo proposed continuing usability evaluation after deployment using ***remote user testing*** (Hartson and Castillo 1998). An advantage of testing remotely is that usability data is more representative of real world usage. However, it requires some kind of remote capture method. They developed a method based on real users reporting critical incidents (Hartson and Castillo 1998). Since then, several tools have been created to automate some stages of remote user testing, and with the advent of the web they are mainly based on log analysis (Paganelli and Paternò 2003; Fernandez et al. 2011). Paternò has been actively working in this area for some time. In 2003, Paganelli and Paternò propose using remote user testing within a model-based evaluation (Paganelli and Paternò 2003). An expert evaluator must first create a task model using a task tree editor. While performing the remote test, these tasks are displayed in the browser for the user to select, and their logging tool gathers events for the selected task. After the tests, their tool called WebRemUsine provides visualizations of results like performed tasks, errors, and loading time. Later on, the tool Web

Usability Probe (WUP) was developed with more sophisticated analysis and visualization tools using timelines to display user interaction events of different kinds (Carta et al. 2011; Burzacca and Paternò 2013). The task model is relaxed to a list of tasks with precedence dependencies, which the designer must execute first to feed WUP with the optimal use for each task (without errors and useless actions). WUP will then compare user logs with the optimal log and display their timelines with the intention to help designers identify potential usability problems in the deviations between timelines (Carta et al. 2011). A similar tool is WELFIT, where event logs are visualized through usage graphs (Santana and Baranauskas 2015). Similarly to WUP, it requires an expert to find deviations in walks present in the graph, which they call usage incidents, although it provides additional help by summarizing events in usage patterns. Moreover, in general these methods are focused on supporting the UX expert during the analysis of problems, but not on supporting developers during the implementation of solutions for those problems. Besides, nothing is mentioned about what happens if the solutions fail. To overcome these problems, we propose: first, to support the easy implementation of solutions by means of CSWRs and second, to use A/B testing for usability evaluation. Thus, UX experts may create and test different alternative solutions applying CSWRs without involving developers until the best solution is found.

In a similar way that code refactorings apply changes to solve problems with the internal qualities of code (called code smells) (Fowler 1999), ***web refactorings*** may be used to solve problems with the external qualities of code, like usability or accessibility (Garrido et al. 2011). We have developed a catalog of refactorings to improve usability, called *usability refactorings,* and each refactoring is linked to the usability problem that it may solve, which similarly to code smells, are called *usability smells* (Garrido et al. 2011). We have also conducted an experiment to demonstrate the real gain that usability refactorings produce on effectiveness in use, efficiency in use and satisfaction in use (Grigera et al. 2016). Moreover, we have found that many usability refactorings may be applied *at the client-side* (through CSWR), which lowers the cost of creating different solutions substantially. Thus, we have developed a framework that allows applying CSWRs, thus reducing the time to implement the solutions and reducing the overall cost of changing a running system at the server (Garrido et al. 2013c).

While creating the catalog of usability refactorings and experimenting with them, we have found that there may be several refactorings applicable to solve the same usability smell (Distante et al. 2014). This fact, together with the need of testing different alternative solutions to find the best one as we mentioned above, make the use of ***A/B testing*** very appealing. A/B testing methods are a form of online controlled experiments (Kohavi and Longbotham 2015). The simplest setup of an A/B test is to have users randomly exposed to one of two variants of one factor: Control (A), which is normally the default version, and Treatment (B), which is the change to be tested. When there is more than one treatment it is also called *split testing*. The value of A/B testing is that it allows establishing a causal relationship between a change and its effect on user-observable behavior (Kohavi et al. 2009). As a notable example, having at Amazon a system that made running experiments easy allows the company to innovate quickly and effectively (Kohavi et al. 2009). It also has the added advantage of lowering the cost of user testing in the sense that it does not need to recruit and pay testing subjects.

2.2 Related Work

Although A/B testing is specially used by companies to increase revenue, it may as well be used to evaluate usability. In 2014, Speicher et al. pointed out that usability evaluation is only applied at very slow iteration cycles in today's industry, in contrast with the efficiency and easy-to-deploy nature of A/B testing (Speicher et al. 2014). Thus, they proposed using A/B testing in the context of a UEM that they have specifically used to detect usability problems in search engine interfaces (Speicher et al. 2014). Analogously, AttrakDiff is a tool available on the web that allows testing the user experience (UX) of a web application before and after implementing changes in it, thus, applying A/B testing to detect the effects of the changes on UX (UID 2018). Particularly, their method is based on a questionnaire that the tool presents to users to measure the hedonic and pragmatic quality of the application (Hassenzahl 2006). In contrast, our method does not limit the kind of web pages or usability problems that may be found, and our tools allow measuring different aspects of user interaction (besides customer satisfaction), like the time to complete a task (efficiency), task completion (effectiveness) or recording user interaction events.

We also advocate the use of an **iterative process** of usability testing and improvement as a web application grows and changes. An *iterative usability process* has been defined by Genov as a way to obtain continuous feedback (Genov and Alex 2005). According to Genov "…Experimental rigor is not necessary for all usability research/testing. There are evaluations whose main goal is to help guide interface design and development, namely iterative or exploratory usability testing" (Genov and Alex 2005). The same author claims that it is relevant to test with a few users in order to obtain feedback and determine usability problems without the need of big experiments involving hundreds or thousands of users in order to accept or reject a null hypothesis. It's worth to note that Genov proposes a general scheme for applying control system theory to iterative usability testing at early stages of design. There is also related work that shows how an iterative usability testing process was applied successfully in the context of a Library website (George 2005). However, they performed ad-hoc usability evaluation for a very specific domain while we propose a method that could fit on any website, and also full supported by a specific tool at each stage.

Among the approaches aimed at incorporating usability at early stages of development there are proposals to integrate *Functional Usability Features* (FUF) as primitives in Model Driven Development (MDD) methods (Panach et al. 2015). There are also several efforts to combine User Centered Design (UCD) with agile methods, since both processes are user-centered and iterative (Silva da Silva et al. 2011; Jurca et al. 2014). Agile methods propose an iterative process of development involving frequent design evaluation and redesign, centered on user feedback (Williams and Cockburn 2003), although they do not necessarily address the usability of the product, which is the specific goal of UCD (Jurca et al. 2014). However, in contrast with agile methods, UCD spends a considerable effort on analysis and specification before development begins, it is based on up-front design, and uses inspection methods to evaluate early prototypes within longer iterations than a typical agile sprint, so the integration of both approaches is not trivial. The main challenges are to synchronize their activities and practices, which usually depend on the particular context of the company, and understanding the roles played by UX experts. Proposed practices include using textual scenarios to communicate the use context to the development team, in agreement with the customer (Obendorf and Finck 2008); having a Sprint 0 where experts focus on the UX design (Da

Silva et al. 2013), performing *Little Design Up Front* (LDUF) and keeping UX experts *One Sprint Ahead* of development (Silva da Silva et al. 2011). However, in general UX experts are too overworked to keep up with this practices, usually working on several projects at a time and in many stages of each project, lacking tools to automate the activities (Jurca et al. 2014).

In this work, instead, we aim to apply an iterative method of usability improvement once there is a user interface (UI) for some functionality that can be exercised through user testing and that can be incrementally improved, i.e., a Minimum Testable Product (MTP). The difference between an MTP and an MVP (Minimum Viable Product) is that the first allows for earlier and faster iteration (Schissel 2014). In this context, Lee and McCrickard proposed scenarios as a tool that can go along with the entire development cycle, from requirement analysis to usability evaluation after a release (Lee and McCrickard 2007). Shortcomings of the approach include the overhead for developers to maintain scenario design documentation updated as development progresses, and the cost of manual evaluation of usability using walkthroughs. Similarly, Benigni et al. propose that experts perform usability evaluation manually using cognitive walkthroughs with little involvement by end-users (Benigni et al. 2010). Moreover, Detweiler recommends conducting design reviews with UI designers early and often, and wait until after product release to conduct user tests, favoring remote usability testing when end users are difficult to reach (Detweiler 2007). However, his work only provides tips but, to the best of our knowledge, there are no concrete proposal in terms of methods or tools to conduct user testing capturing real usage data timely to improve usability in subsequent iterations of an agile process.

3. ITERATIVE USABILITY IMPROVEMENT METHOD

We propose an iterative and incremental method of usability improvement for web applications that integrates A/B testing and usability refactoring, and may be used in the context of any development process, even an agile process. Each iteration or increment in our method is aimed at evaluating and improving the usability of a small set of tasks. To simplify the description of this method, we will assume that the set of tasks evaluated in an iteration are all related to a particular use case. Besides, testing a use case at a time will provide a fine-grained analysis of the problems. In Section 5 we will come back to this discussion in the context of an agile development process.

The method is divided in five main stages that range from test case specification to discovering and implementing the best solution for each problem found during testing. Figure 1 depicts the method and its stages, with arrows showing the sequence of stages and forming a cycle to underline its incremental essence. The five stages composing the method are:

1. Usability test specification and execution
2. Results analysis, design and assembly of alternative versions
3. Scenario specification and test execution for each application version
4. Results analysis and identification of best version
5. Refine best version implementation

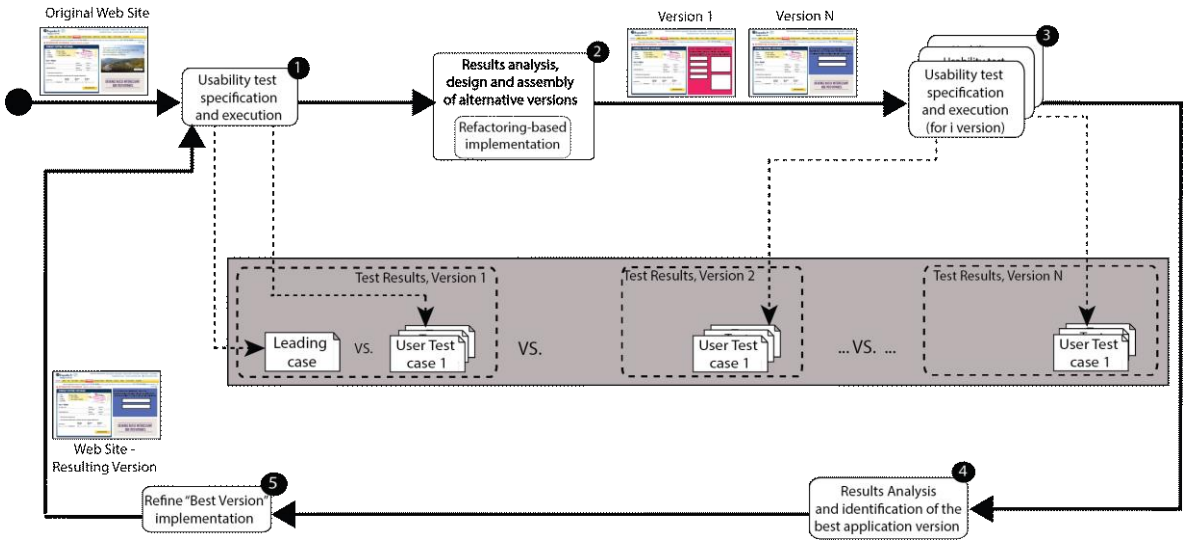


Fig. 1: Iterative Usability Improvement Method

The rest of this section provides first an overview of the stages in our method, naming the models and the classes in each model that get instantiated at each stage. The second subsection describes the two metamodels that support the usability improvement method and define the classes used at each stage. In the description, we distinguish three **roles**: evaluator, test subjects and developers. We call *evaluator* to the person that will use our method and tools to define and conduct the usability tests for the web application, commonly, a UX expert. *Test subjects* or simply *subjects* are recruited by the evaluator to run the test cases remotely under a usability test. *Developers* are in charge of implementing the best or winning version at the main application in the last stage.

3.1 Method Overview

We next describe briefly the steps involved at each stage of the method.

1. Usability test specification and execution: at this stage the evaluator first selects a particular use case to be tested in this iteration, and defines a **UsabilityTest** for it. When defining a **UsabilityTest,** the selected functionality is decomposed into **Tasks**; for example, to test "Buy products", we may define **Tasks** such as "Create a new user account", "Login", "Search for products", "Add products to cart" and "Check-out". Defining these **Tasks** allows the evaluator to control the test and analyze the results in smaller units of interaction rather than in a single block. For each **Task**, the evaluator must follow the sequence of steps (i.e. the user interaction with the UI) for performing them in the current version of the application. This implies defining the test **Scenario** by example. While doing it, the evaluator must associate the corresponding subset of interaction steps to each **Task**, creating what we call a **Task4Version**. Thus, a **Task4Version** represents the interaction steps required for performing a **Task** in a particular application version. While instantiating these classes, the evaluator will be creating a *Scenario Model* for the test. Then, the evaluator should describe the measurements that the tools will track while the test case is executed, creating an *Observation Model*. Thirdly, the evaluator may execute the test herself to create the *leading case*, also known as optimal or pilot case in related works, i.e., the case that provides the best results for the observations defined in the Observation Model. Finally, the test is executed by test

subjects, and the results of each observation are recorded. Thus, this stage produces test results corresponding to a Scenario Model of the original site.

2. Results analysis, design and assembly of alternative versions: this second stage has the purpose of analyzing the results of the 1st stage to discover usability problems and to create alternative application versions as solutions for those problems. Problem discovery may be accomplished manually by visualizing test results or automatically with the help of tools, as will be described in Section 4. Once problems were identified, we propose matching them with catalogued *usability smells* (Distante et al. 2014; Grigera et al. 2017). Examples of usability smells are "Abandoned form", "Undescriptive element", and "Misleading link" (see Appendix A). The benefit of matching problems with known usability smells is that each smell points to its solution in terms of a *usability refactoring*, and most of them may be applied semi-automatically at the client side (as CSWRs). In some cases, there may be more than one refactoring suggested as solution to a given smell. Nevertheless, this is not a drawback but an opportunity to try out different ideas and discover the one that most users prefer or find more usable. Thus, the evaluator herself may design several application versions by combining different CSWRs for each problem.

3. Scenario specification and test execution for each application version: in the third stage the evaluator should define test scenarios for each application version. The tests will exercise the same **Tasks** than the 1st stage, and compute the same measures, so this stage and the 1st share the same Observation Model. However, each new version may require a different **Scenario** and related **Task4Versions**, as the particular steps of the tasks may differ from one application version to the other.

Similarly to A/B testing, the evaluator should split the number of test subjects assigned to each application version, and have them run each test case remotely.

4. Results analysis and identification of best version: at this stage, the evaluator should analyze the results of the test execution in the 3rd stage. Since all test executions collect the same metrics (those defined in the shared Observation Model), the evaluator may compare how each version performed in terms of each metric, even comparing with the results of the 1st stage, or in terms of a particular refactoring, and single out the version that performed best, identifying also the best refactorings.

5. Refine best version implementation: once the best version for the particular task or use case is identified, the remaining step is to implement the solution at the server. We do not provide the implementation of refactorings at the server because the mechanics of the transformation depend heavily on the technology used. However, since the mechanics involved in CSWRs apply simple and mainly small changes, developers should find it easy to implement.

After the fifth stage, the process starts again with a new usability test for a different functionality.

3.2 Metamodels that Support the Process of Usability Improvement

Our approach aims to test and improve how users perform specific tasks when using a Web application, and try different scenarios for each task to discover the best version. This is

similar to say that we aim to improve *"usability in use"*, which is defined as "the degree to which specified users can achieve specified goals with effectiveness in use, efficiency in use and satisfaction in use in a specified context of use", according to ISO/IEC 25010 (ISO 2011). This method of usability improvement is fully supported by two metamodels:

- **Scenario Metamodel**: it allows defining the specific end-user tasks involved in a usability test, as well as the concrete steps that compose a task in each particular version of the application. It also includes the CSWRs that are applied to obtain each application version.
- **Observation Metamodel**: it allows specifying the metrics to be calculated during the execution of a particular task being tested, which may be related to effectiveness, efficiency, satisfaction, and also to user interaction events that are of particular interest to the evaluator.

The rest of this section provides a precise description of each of the above metamodels.

3.2.1 Scenario Metamodel

Our approach is a controlled form of remote usability testing where test subjects execute a guided, predefined sequence of steps, which is called **Scenario**. The purpose of having all subjects test the same **Scenario** is to ease the comparison of results among **Scenario** instances for the evaluator to identify usability problems in an application. The steps that compose a **Scenario** are called **InteractionSteps**, and each one represents any kind of interaction like follow a link, fill an input form, submit a form, etc. Thus, **InteractionSteps** are highly tied to a particular application version. Since it could be that different kinds of interactions require distinct data, **InteractionSteps** have a collection of properties modelled with the **Attribute** class.

Meanwhile, we define the concept of **Task** as an abstract unit of test. A **Task** represents a cohesive group of interactions that are part of a bigger functionality, which would be the one specified in a use case or user story. Imagine the following simple user story, which could be specified in the context of an e-learning web system.

> *As a student, I want to submit my class work solution files individually so I can introduce a specific description for each of them.*

From this simple *functionality*, the evaluator may specify, for instance, three component **Tasks**: "User Login", "Search and Select Course", "Describe File and Upload".

Thus, in the first stage of the usability improvement method, the evaluator defines the **Tasks** to test in the current round, and the same **Tasks** are tested in the third stage, in order to compare which version of the application provides the best test results for those particular **Tasks**. It is important to note that these **Tasks** are independent of any particular application version. Each different application version may involve a different sequence of **InteractionSteps** to perform these **Tasks**. With this in mind, we define the concept of **Task4Version** as the mapping between a **Task** and the sequence of **InteractionSteps** that compose the **Task** in a particular application version, i.e., a particular **Scenario**. Therefore, the model generated allows evaluators to establish a mapping between the different versions of a **Task** corresponding to each **Scenario**, thus making it straightforward to compare the performance of a **Task** in all its versions.

To create each application version, we propose the application of client-side web refactorings (CSWRs). As previously explained, a CSWR is a software component aimed at solving a usability smell in a particular way by modifying the application's web interface on the client-side. The Scenario Metamodel allows specifying the **Refactorings** that were applied to generate a particular version of a **Task**, i.e., a **Task4Version**, thus allowing to track the performance of each refactoring during the tests. All previously defined concepts and their relations are depicted in the Scenario Metamodel diagram shown in Figure 2.

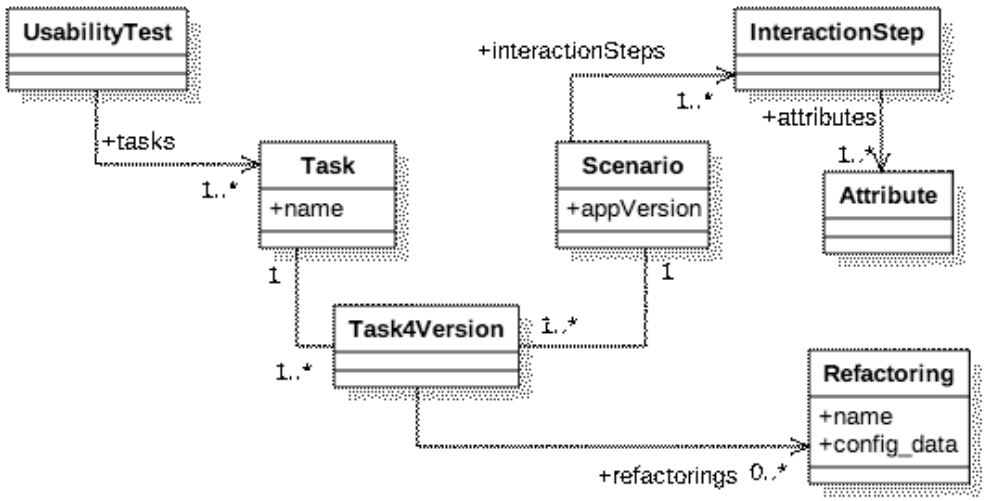


Fig. 2: Scenario Metamodel

3.2.2 Observation Metamodel

Once the Scenario Metamodel is instantiated to create a usability test, the evaluator must specify the data that should be collected and "observed" to measure and compare test executions. With this purpose, we defined the Observation Metamodel, which is based on the Observation Pattern (Fowler 1997; Yoder et al. 2002). This pattern defines two kinds of observations: **Measurements** can have any numeric value within a range and possibly an appropriate unit, like the measurement of "time" in "seconds", while **CategoryObservations** contain a value within a finite discrete set, such as "satisfaction" in some discrete scale. The class diagram for the Observation Metamodel appears in Figure 3.

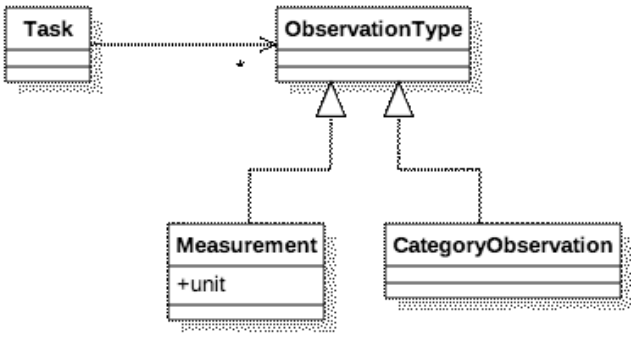


Fig. 3: Observation Metamodel

We now show how to use this metamodel to measure *usability in use* as stated at the beginning of Section 3.2, which includes *effectiveness in use*, *efficiency in use* and *satisfaction in use* (ISO 2011). ISO/IEC 25010 defines *effectiveness in use* as "the degree to which specified users can achieve specified goals with accuracy and completeness in a specified context of use". According to this definition, to measure *effectiveness in use* we need the percentage of success obtained per completed task (accuracy and completeness). Thus, we may define a **CategoryObservation** with two possible values: *1*, which means that the activity was finished successfully; and *0*, which means that the activity failed.

Meanwhile, *efficiency in use* is defined as "the degree to which specified users expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use" (ISO 2011). According to this definition, to measure *efficiency in use* we need the subjects' time to perform a task, considering the reached level of effectiveness. In this case, we may define a **Measurement** of completion time in seconds and calculate the effectiveness-time ratio.

Lastly, ISO/IEC 25010 defines *satisfaction in use* as "the degree to which users are satisfied in a specified context of use" (ISO 2011). According to this definition, to measure satisfaction we need the subjects' personal opinion about their perception while performing each task. For this we may define a **CategoryObservation** with a satisfaction question which takes for an answer a value in a 5-point Likert scale: "Totally agree", "Fairly agree", "Undecided", "Fairly disagree" and "Totally disagree". These questions appear at the end of the task in which the observation is defined.

Altogether, when both metamodels are instantiated for a particular usability test, the resulting models are flexible enough to specify test scenarios comparable among them, as it is illustrated in Figure 4. This is possible given that, although the Observation Model is defined by using **Task**, each of these **Tasks** is mapped to different **Task4Versions**, in correspondence with each application version.

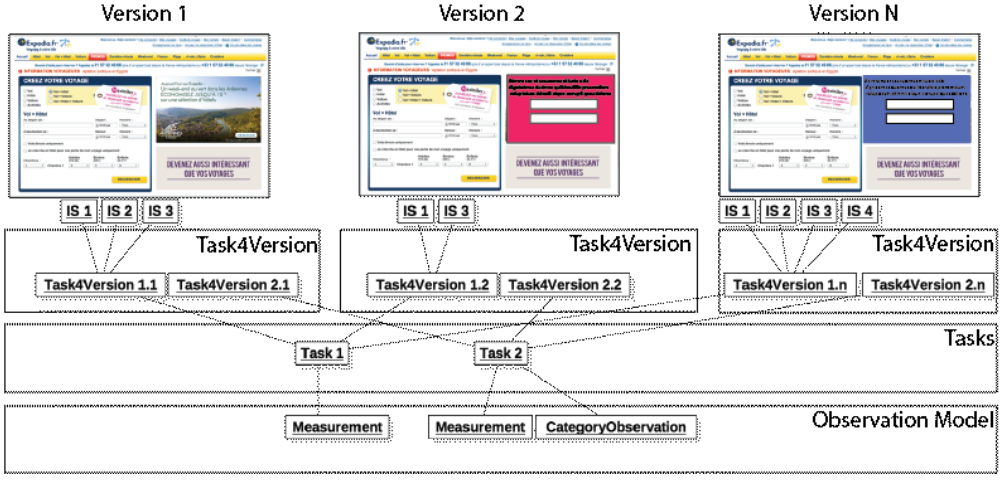


Fig. 4: Relationships between models

4. SUPPORTING TOOLS

The method proposed in this article uses different tools specifically designed for supporting the evaluator at each step. The first subsection presents our toolkit, describing how it is used at each stage of the method. The rest of the section explains the tools in detail. While it is not mandatory to use these tools to apply our method, they empower the evaluator in a unique way by: easily defining test scenarios by example, running tests remotely with automatic metric gathering, creating different application versions by applying CSWRs semi-automatically, and visualizing test results that include traces among the different application versions and the refactorings applied to each one. In consequence, these tools contribute in making our method feasible and practical in the context of an agile development.

4.1 Tools at Each Step of the Method

In this section we refine the previous overview, identifying each step of the process together with the supporting tools specifically defined for each of them. Figure 5 shows the fine-grained listing of the steps at each stage (a refinement of Figure 1), side by side with the supporting tools. Note that this figure depicts the use of the toolkit for a single iteration. In a few words, we have created a platform to define and execute test cases together with an application for visualizing test results, and a framework for implementing usability improvements on the client-side.

In the **first stage**, as described in the overview, the evaluator must create the Scenario and Observation Models for the test cycle. The Scenario Model is first created by example (**Point 1.a** in Fig. 5). Our tools allow for this model to be recorded automatically and later refined. This is done by example, i.e. by interacting with the website while using a tool called ***Scenario Recorder (ScRec)***. The *ScRec* tool creates a **Scenario** specification containing the sequence of all the **InteractionSteps** with the information related to every event triggered by user interaction. This allows the evaluator to define which **InteractionSteps** belong to a particular **Task**(s) that are the subject of the current testing cycle (**Point 1.b**). This point is supported by the ***Scenario Editor (ScEdit)*** tool. Using it, the evaluator may edit the groups of **InteractionSteps** composing each **Task4Version**, add new ones, edit name and attributes, etc. The third step consists of defining an Observation Model (**Point 1.c**) with the help of *ScEdit*. For example, a **Measurement** of *time* can be defined to measure the time spent during a **Task** or a **CategoryObservation** may be defined to add a satisfaction questionnaire about the perceived difficulty after performing a **Task**.

The Scenario and Observation Models, together composing a **UsabilityTest** model, are fed into the ***Scenario Player (ScPlay)*** for it to guide the test execution. In this way, *ScPlay* is aware of the **InteractionSteps** required for a task since it "*plays*" a previously defined **Scenario**, showing each step to the user while it "*observes"* the current interaction, recording the metrics defined in the Observation Model. At this point, the evaluator may use *ScPlay* herself to create the leading or *optimal case* (**Point 1.d**). That is, the evaluator interacts with the website again but this time *ScPlay* will record the metrics defined and consider them as the best or optimal metrics. This step is optional. Next, all test subject will execute the test using *ScPlay* with the same specification of **UsabilityTest** (**Point 1.e**). All test results are stored on a database. All these three tools, *ScRec*, *ScEdit* and *ScPlay* are deployed as a Web Browser plugin.

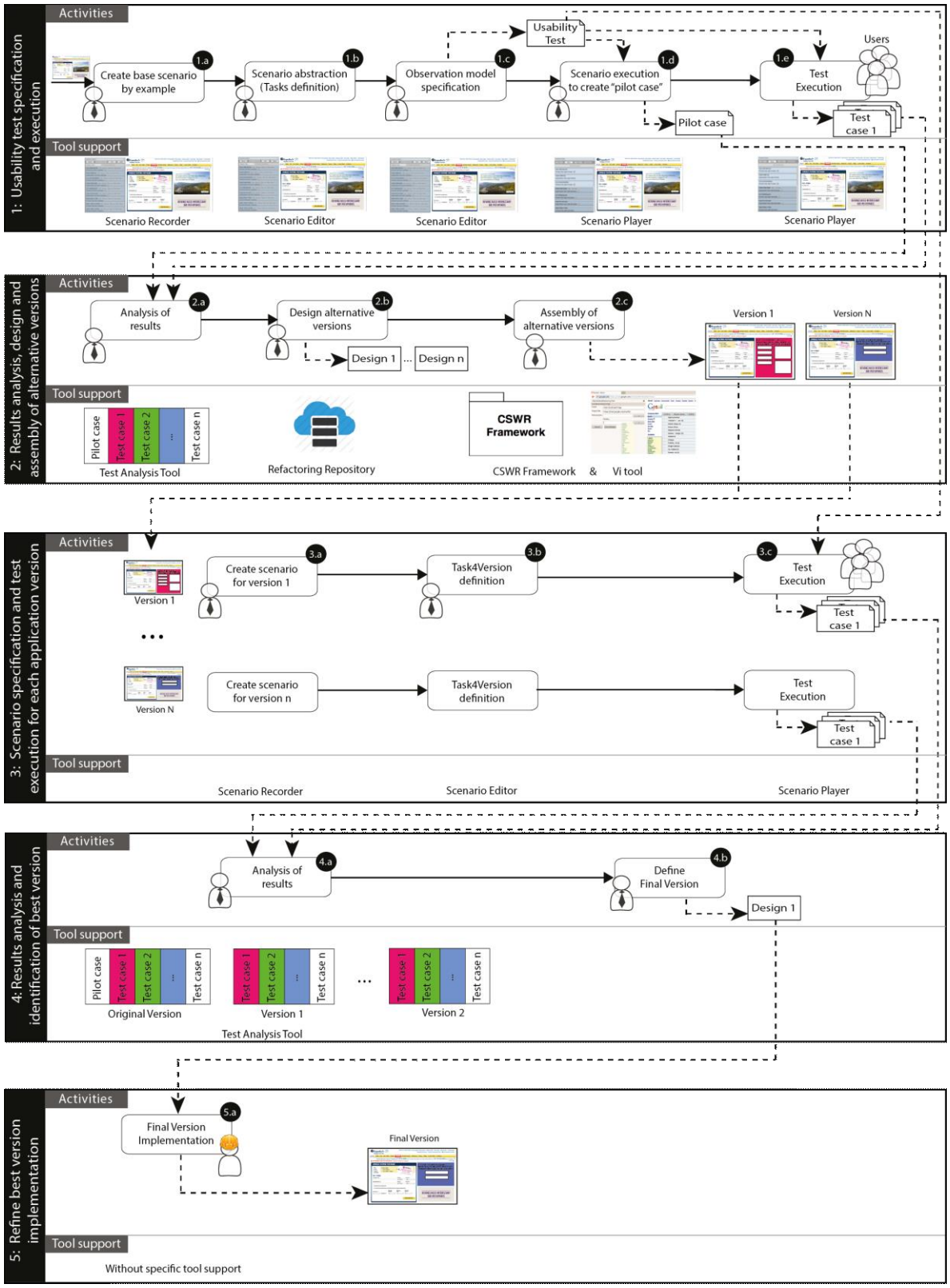


Fig. 5: Method stages and tools supporting each step

In the **second stage**, the evaluator analyzes test results. This activity is supported with a Web application called ***Test Analysis Tool*** (called ***VisualA***), which provides different visualizations for test results (**Point 2.a**). If the evaluator recorded an optimal case, usability problems may be identified in deviations between test results from subjects and the optimal results, as suggested by other approaches (Carta et al. 2011). Moreover, the evaluator may review available literature on usability guidelines and catalogs of *usability smells* (Distante et al. 2014; Grigera et al. 2017).

In a parallel project, we have constructed a tool called USF that is able to identify usability smells automatically by analyzing user interaction events (Grigera et al. 2017). However, this tool requires a critical mass of users to work properly. If the evaluated application has been released, then it would be possible for the evaluator to use USF to identify usability smells and to analyze user interaction events at a higher level of abstraction. Note that USF does not require the specification of a usability test with tasks, but on the contrary, the tool captures usability smells "in the wild", while users interact with the UI.

The advantage of identifying problems as usability smells is that they are bound to specific solutions in terms of *usability refactorings*, and most of these can be applied in the client. Moreover, in some cases there are alternative refactorings that may solve the same smell, and this provides an excellent context to try different solutions as in A/B testing.

Thus, the following step consists on designing new application versions by combining usability refactorings (**Point 2.b**). To implement these new versions, the evaluator may browse the ***Refactoring Repository (CSWR Repository)*** searching for specific CSWRs to solve each problem (**Point 2.c**). In this repository, there are abstract specifications for each CSWR, which can be easily instantiated by extending the ***Refactoring Framework (CSWR Framework)*** to create a concrete refactoring for a target website. This instantiation may be done by using the ***CSWR Visual Instantiation Tool (CSWR VI Tool)*** (Garrido et al. 2013b), that allows non-skilled users to create refactorings. All instantiated CSWR are saved in the repository, and will be automatically executed during the tests by *ScPlay*.

For each application version, new tests cases are obtained by repeating the Points 1.a–1.e for each version (**Points 3.1**, **3.b**, **3.c** and **3.d**). Test subjects are divided among the different versions and each group executes the assigned test (**Point 3.e**). When all the test cases are stored, the evaluator uses again the *VisualA* tool to compare different application versions and identify the best solution (**Point 4.a** and **Point 4.b**). Finally, the best solution should be implemented natively by modifying the application code (**Point 5.a**). Since this step depends strongly on the underlying technological support behind the Web application, we do not provide specific tool support.

In the following subsections we introduce some details about the tools that support the automation or semi-automation of the usability improvement process.

4.2 Scenario Recorder and Editor

We have developed a tool (which runs as a Web browser extension) that allows evaluators to define their scenarios. The tool is called ***Scenario Recorder*** (*ScRec* for short). This tool for creating scenarios empowers evaluators with the possibility of recording any UI interaction with a website, thus easily creating the base scenario specification by example. In this way, while other approaches avoid having task models because it's time-consuming to define them (Carta et al. 2011), our method has the added benefits provided by a complete underlying test model, which is easy to build and it allows for better automation and guidance for test subjects according to the evaluator needs.

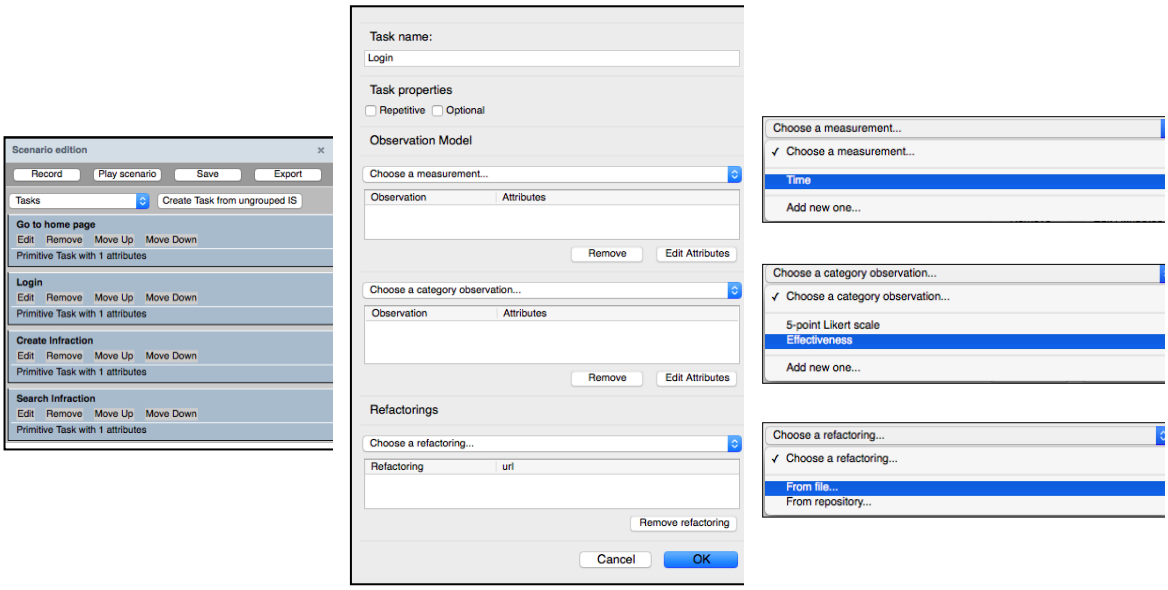
The *ScRec* tool appears at the left of Figure 6, where the **InteractionSteps** performed by the evaluator appear as blue boxes, while she is interacting with the website. Each type of event (mouseover, focus input, etc.) has an associated **InteractionStep** box. Also, every **InteractionStep** may be deleted after the recording. The evaluator may also delete steps to provide more freedom to test subjects to complete a subtask without guidance.

InteractionSteps may be grouped into a **Task4Version** at the same time the evaluator is recording the scenario. In order to do it, while performing the scenario and recording the **InteractionSteps** the evaluator may define a breakpoint to indicate that all **InteractionSteps** performed to that point correspond to a particular **Task**. This is done by clicking the "Create Task From ungrouped IS" button. This functionality groups all those interaction steps performed from either the beginning or the last **Task** created.

a) Left panel with InteractionSteps recorded so far **b)** Zoom in the left panel

Fig. 6. Scenario Recorder

Thus, *ScRec* allows the evaluator to define (by recording) the scenario based on **InteractionSteps** (such as click on form input, fill input, etc.) and to group steps into **Tasks** (such as "Login", "Look for flights from Buenos Aires to Calafate", etc.). To specify other details of the **Task** such as its Observation Model, the evaluator may use the tool called ***Scenario Editor*** (*ScEdit* for short). This tool is shown in Figure 7.



a) Task Model **b)** Task edition. **c)** Edition of a single task

Fig. 7. Scenario Editor: Observation Model and Refactorings definition

The reader may note that while in Figure 6 the sidebar panel shows the **InteractionSteps** (because it is the selected view), in Figure 7.a the panel shows the list of defined **Tasks** instead. Moreover, for each **Task** listed in Fig. 7.a, an "edit" button is shown; by clicking it the evaluator may configure the Observation Model for each **Task** (among several aspects that appear in Fig. 7.b). Observations to a **Task** may be added as Fig. 7.c shows. The top dialog shows the selection of a **Measurement** for the abstracted task "Login", for instance to measure the time used by participants when running this scenario. Meanwhile, the middle dialog in Fig. 7.c shows the selection of a **CategoryObservation** for example to add a satisfaction question for the selected task.

For defining different application versions while using the Web application, the evaluator may also specify which refactorings to apply to generate a new version (shown in Figure 7.c bottom dialog). Finally, the evaluator may save the specification and continue with the process. That specification may also be exported as an XML or JSON file and imported again later to be edited or played.

4.3 Scenario Player

Once the **UsabilityTest** is completely defined, the next step is to run the test. For this purpose, the tool called ***Scenario Player*** (*ScPlay* for short), which is part of the same Web Browser extension, allows importing the test and run it. Running the test just means using the Web normally while *ScPlay* collects all the observations defined in the scenario at the same time that the user performs the specified tasks. The *ScPlay* tool provides two visualizations: one that shows a left panel with the list of interaction steps in the current task (similar to the one in Fig. 6.a but without the possibility to edit steps), and another that instead of a left panel, shows a bar at the bottom with only the information about the current step being executed. This second view of *ScPlay* appears in Figure 8. The bottom bar shows, from left to right: a

button for users to switch to the other visualization, the number of interaction steps left to execute in the current task, the instructions for the current interaction step, and control buttons “Skip” (to skip to the next activity), and a red “x” (to quit the task). We allowed skipping single steps to prevent users from aborting a full task just because of a single unfinished step. Then, if effectiveness and/or efficiency are part of the observations, they are calculated as the proportion of successful vs. unsuccessful/skipped steps.

For some interactions steps, *ScPlay* is able to recognize if the step has been completed, thus automating the measure of effectiveness. For example, it can check whether a navigation-based activity is successful by checking destination URL patterns, or asserting the presence of specific UI elements. In the cases when this automation is not possible, the bottom bar shows an additional green button called “Done”, which users should click to manually indicate completion of the current activity.

Using the *ScPlay* tool, the evaluator may generate the leading or optimal case, as soon as she finishes specifying the test. The same tool will be used by test subjects to exercise the test, which can be done remotely in their own browsers.

When the scenario is finished (usually when the user performs the last **InteractionStep** defined in the Scenario), the tool reports automatically the results to a server, along with the user’s related information.

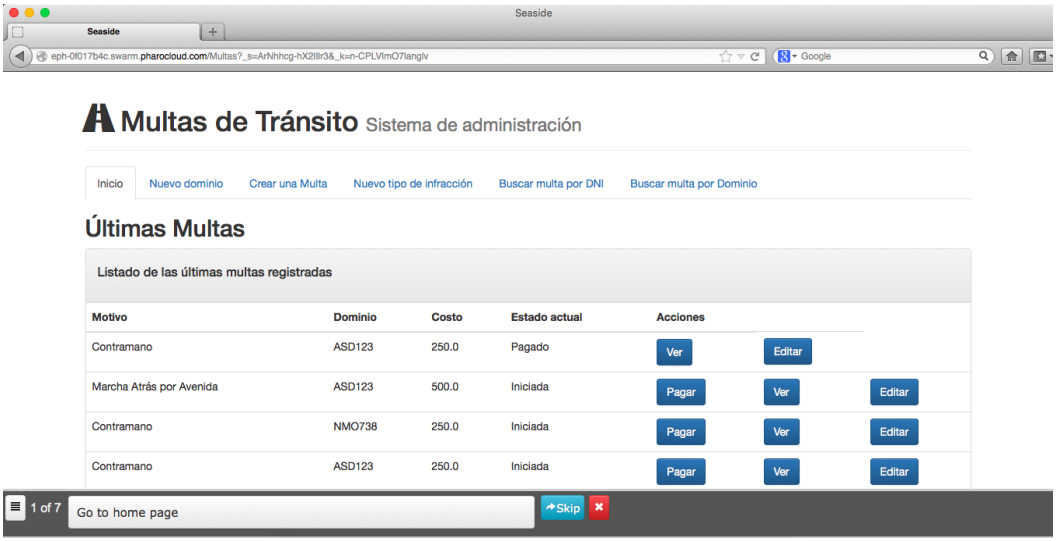


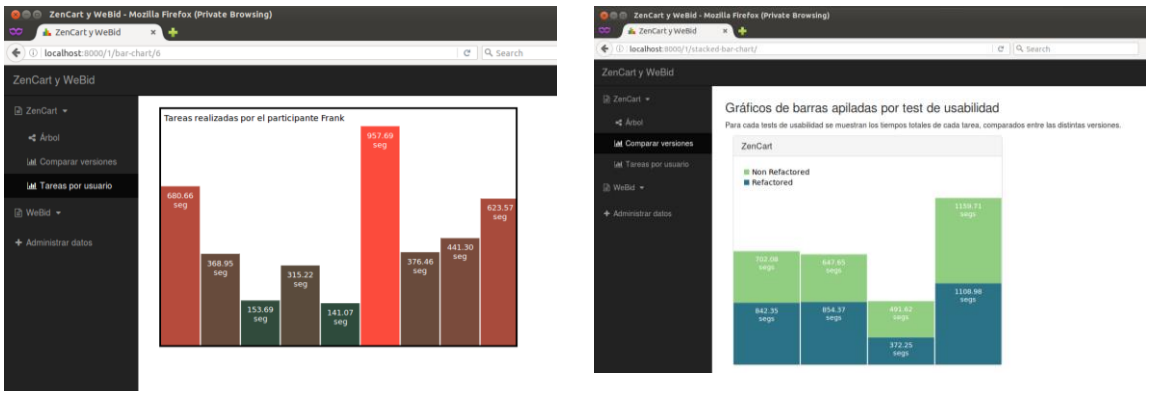
Fig. 8. Scenario Player

4.4 Test Analysis tool

As soon as **UsabilityTest** results are uploaded to a server specified by the evaluator, the ***Test Analysis Tool*** (*VisualA* for short) will show the results in different charts and diagrams. The *VisualA* tool is a Web application which deployment URL is configured as metadata of the scenario specification.

The evaluator may analyze specific aspects of each particular test case, and she may also compare test cases among users or application versions. The main idea is to offer

visualizations of results focused at different levels of detail. For instance, Figure 9.a shows all the **Tasks** performed by a specific user. In this visualization, the time is used as variable of analysis for a single test case. However, results may also be analyzed in a more coarse-grained way. Figure 9.b shows the comparison between each **Task** corresponding to two different application versions. In this example, the focus is on the average time for each version.



a) Tasks performed by user b) Task performance for each version

Fig. 9. Test Analysis Tool

Detailed views of the usability tests are also available. A tree visualization offers to the evaluator a detailed view for each application version. The root node of the tree (depth 0) represents the **UsabilityTest**. The nodes at depth 1 represent the different application versions. For each of these versions, the nodes at depth 2 represent each **Task4Version**, while the nodes at depth 3 are the **InteractionSteps** composing them. Figure 10 shows the tree visualization. For each **Task4Version**, the evaluator may inspect the **InteractionSteps**, together with the values obtained for the Observation Model. At the right of Figure 10, a zoomed image shows summarized information related to the time spent for a specific **InteractionStep**. The same information is available for each **Task4Version**.

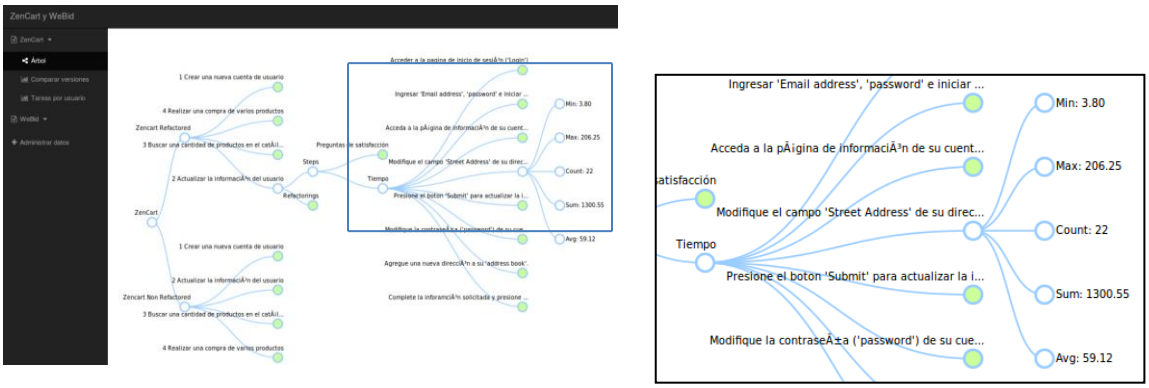


Fig. 10: Detail View for comparing Test Cases

Also for each **Task4Version**, the evaluator may inspect the **Refactorings** applied to this version. This appears in Figure 11, with a zoomed image at the right.

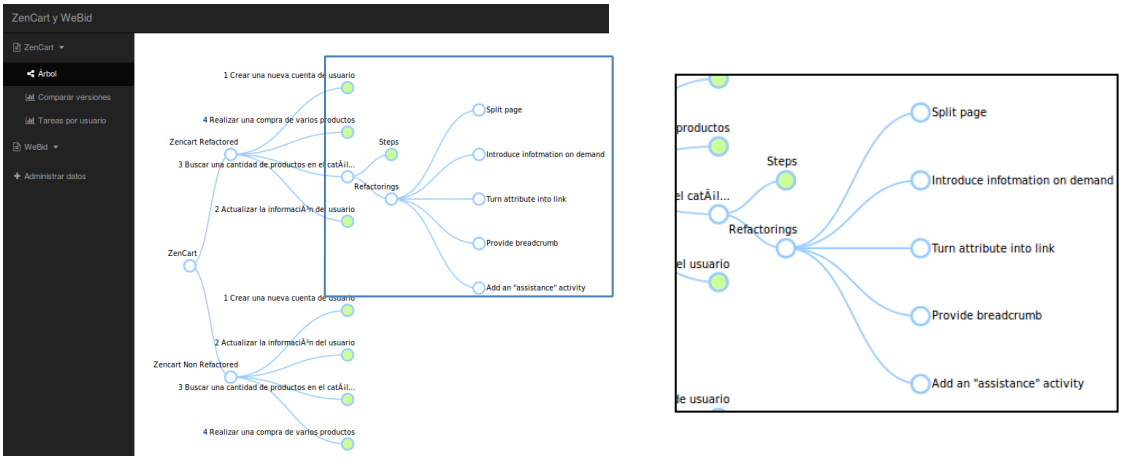


Fig. 11: Detail view of the refactorings applied to a specific Task4Version

Altogether, the evaluator may visualize very specific information (for instance the time spent by a specific user for each **InteractionStep**), processed information (such as the average time for each **Task** for a given application version), or even comparison views (as comparing the average time used for each **Task** in different application versions). Note that although most of these examples show the time as a variable for analysis, other variables defined in the Observation Model could be used.

4.5 Client-Side Web Refactoring Framework and Repository

Our toolkit includes a JavaScript (JS) framework for applying CSWRs to create new application versions and test these versions before adhering to a specific one in the server code. That is, the server application is preserved and different versions are created by installing CSWRs as scripts in the client browser. For example, a usability smell like "Free input for limited values" in a form may be solved by applying CSWRs *"Add Autocomplete"* (A) or either *"Turn text field into select"* (B) introducing a drop-down menu (see Appendix A). Meanwhile, the same form may have the problem of "Unformatted date input", which could be solved by a refactoring like *"Add date picker"* (C) to replace it by a calendar widget, or either *"Format input"* (D) to add automatic formatting. Thus, new application versions could be easily created by applying refactorings A and C, A and D, B and C, only B, etc.

To create different application versions for the purpose of testing there are mainly two sets of supporting tools: the ***CSWR Framework & VI tool*** and the ***CSWR Repository***.

The *CSWR Framework* adapts existing applications at the client-side by changing their DOM structure. CSWRs are generic configurable scripts that perform alterations on a webpage's DOM. CSWRs are instantiated by providing specific parameters like the target webpage and specific webpage's DOM elements. This instantiation of a generic CSWR may be done programmatically, or using a visual tool. In the first case, a JS programmer extends the framework writing code that specifies the parameters to instantiate the generic script. In the second case, the evaluator without any programming skill may use the ***CSWR Visual Instantiation Tool*** (*CSWR VI*). This tool that lets the evaluator point-and-click on the target page to select the components that act as values for each refactoring parameter. Further details of the framework and the *CSWR VI* tool appear elsewhere (Garrido et al. 2013b),

although we have recently upgraded it according to the last version of both the Web browser extension technology[1] and our CSWR framework.

Thus, for each usability smell and refactoring selected to solve it, the evaluator may search the *CSWR Repository* which contains the generic versions of each refactoring, and instantiate the selected one using the *CSWR VI* tool. All instantiated CSWRs are also saved in the repository. Note that the set of generic refactorings in the *CSWR Repository* is extensible, though it requires JS programming skills to create a new generic CSWR. The evaluator may ask a developer to create a new refactoring, which may be based on the existing catalogs of usability (and accessibility) refactorings available (Garrido et al. 2013a; Distante et al. 2014; Grigera et al. 2016, 2017). We list in Appendix A the existing CSWRs that exist in the *Repository* in their abstract form, i.e., ready to be instantiated.

Different application versions are achieved by using different combinations of CSWRs or different instantiations of the same one (with alternative parameters). The only restriction to combine CSWRs is that they don't interfere with one another (i.e., they don't apply changes on the same DOM element). Note that each CSWR is a script which applies changes once the target page is loaded on the client-side. Thus, not all possible changes may be applied on the client, but we found that most refactorings can, since they usually produce small, concise changes (Grigera et al. 2017).

In the listing of Appendix A, the reader may see several kinds of improvements that can be applied over existing Web pages. Nevertheless, a very important aspect of our approach is its extensibility. In this point, it is important to consider existing work in the field of Web augmentation, where it was studied that at client-side a huge range of requirements could be satisfied; functional requirements and also non-functional requirements (Diaz and Arellano 2015; Firmenich et al. 2016).

Notwithstanding the fact that the existing CSWRs are enough to perform several usability improvements and also that new CSWRs may be created to tackle new bad smells, it may happen that the UI refactorization power of CSWRs (which is restricted to DOM manipulation) is not enough in some specific cases. For instance, if further information from a database is needed and this information is not easily obtained from the original UI, then a server-side refactoring is necessary. In cases like this, we envision two ways for allowing the UX expert to coordinate it, based on two different dependence levels regarding the development team. First, if the scenario for the test allows it, the UX expert may ask a developer to write a small HTML snippet based on mocked information to be added in the web page (replacing an original DOM element or not) without instantiating a particular CSWR, but directly using that HTML code. Second, if there is no way to do it, a possibility is to involve a developer that specifically creates the refactoring at server-side. It is important to mention that doing it at server-side not necessarily implies to modify the current application release, but that the required server-side process (for instance, to perform a database query before building the corresponding UI) may be encapsulated behind a web service that works independently and returns the UI component for replacing the original one where the usability problem was found. This idea of empowering client-side adaptation with server-side capabilities was already explored in the literature (Urbieta et al. 2017). That work

[1] https://addons.mozilla.org/en-US/firefox/extensions/ (last accessed Jan 17, 2018)

proposes a model-driven approach for defining a web service (using IFML) that generates the UI element to be woven at the client-side. In this way, that new UI element may contemplate complex aspects from the back-end, such as user profile aspects, database queries, etc., but it does not imply modifying the original web application back-end.

5. INTEGRATION WITH AN AGILE DEVELOPMENT PROCESS

Our iterative method of usability improvement, as previously described, applies once there is a Minimum Testable Product (MTP), and we can learn from feedback of testing it. In addition to our method being non-invasive and applicable in the context of any development process, we believe it is especially suited for agile methods since it follows similar principles: shorter design periods, iterative design evaluation, and redesign after feedback from actual usage. Moreover, there seems to be a lack in the literature with respect to methods and tools to integrate usability evaluation in late stages of an agile cycle; instead, as presented in Section 2, the literature is mostly focused on the integration of usability design methods in early stages of the cycle.

The main obstacle that others have found in the integration with agile methods is that usability evaluation usually takes more time than a development sprint, so it is hard to synchronize their activities and practices, and usability improvements become out of synch with development (Jurca et al. 2014).

We propose an integration approach that may be used to synchronize the five stages of our usability improvement method with the events of an agile development process, so that evaluating the usability of a product increment and finding its best version takes no more than one sprint. In the description of the approach, we use the terminology and activities of Scrum, although we try to keep the description as general as possible so other agile methods could also be used (however, details about other methods are out of the scope of this paper). In particular, we assume that the development process starts with the construction of a Product Backlog, which is a list of all the features or user stories that the software product must have, prioritized by value delivered to the customer. Stories are self-contained units of work agreed upon by the developers and the stakeholders, which usually can be tackled by one team member and, when completed, implement a concrete functionality that is directly perceivable by stakeholders. Then, the product is built iteratively in sprints. Every sprint starts with a *planning meeting* to select from the Product Backlog the features for the iteration, breaking them down into tasks to form the Sprint Backlog. Once this backlog is carefully defined, the development team starts working on it in what is called *sprint execution*. A short *daily meeting* is run every workday to share the sprint work progress and new problems found during it. Right after sprint execution and at the end of the sprint, we particularly refer to a Scrum meeting called ***sprint review***. The purpose of the *sprint review* is to inspect and evaluate the product increment built during the sprint (Rubin 2012). Participants include the Product Owner (P.O.), the team, the stakeholders, and even customers. We think this is an appropriate time to do a usability evaluation, as proposed before by Düchting (Düchting et al. 2007).

Moreover, before devising our integration proposal, we conducted a survey with 19 developers of local companies about the adoption of usability practices in agile methods.

The main findings were that most companies have UX experts on the team performing heuristic evaluations as well as user testing to discover UX problems and propose solutions. Also, final users report usability problems in almost 60% of the cases. Most projects use Scrum, and among them, the majority perform *sprint review* meetings at least in some opportunities. Last but not least, 25% of them conduct user tests during *sprint review* meetings, and another 44% think that it's feasible to do it. The complete details of this survey appear in Appendix B. While a statistical experiment demonstrating the practicality and usefulness of our integration proposal is out of the scope of this paper and left as future work, our proposal is compatible with the evidence that we collected from the survey, as well as from the case studies that we conducted which are described in Section 6.

Back in Section 3 we recommended defining one usability test per use case, which usually comprises a set of user stories. Depending on the agile method used, stories from different use cases may be implemented in a cycle. Still, we recommend defining one usability test per use case implemented in a cycle (i.e., launching parallel tests at a time), to gain a better insight of the problems and alternative solutions of small pieces of functionality.

The integration approach that we propose can be summarized as follows:

1) **Initial Sprint**: we call Initial Sprint the sprint planned to have a potentially shippable product increment or MTP at the end, involving a UI that will be evaluated with our method. During the *sprint execution*, team members will have implemented the initial version of some user stories. If the UX expert is part of the team, she may start designing user tests early, even pair programming with a team member to create the tests. Multidisciplinary teams in agile methods are a perfect fit for this (Düchting et al. 2007). Right before the *sprint review*, the UX expert has to finish the specification of the usability test(s), with corresponding tasks and scenarios for this product increment (**Stage 1** of our method). Then, during the *sprint review* meeting, the UX expert chooses some of the participants to run the user test, who can be customers, stakeholders or even the P.O. After the *sprint review*, the expert analyzes the test results to discover usability problems (**Stage 2** of our method).
2) **Second Sprint**: the problems found in Stage 2 are presented to the P.O. and the team during the *sprint planning* of the next sprint, called Second Sprint. For those problems that the P.O. considers important, sprint backlog items (SBI) will be generated for the UX expert to evaluate alternative solutions. During the *execution* of the Second Sprint, the expert designs and assembles alternative versions of the application applying CSWRs by herself, while developers in the team continue with other SBI. The expert designs user tests for each version and starts a new round of testing (**Stage 3** of our method). The participants of these user tests may be customers, stakeholders or real users, performing the test remotely. As stated by Detweiler, running remote usability tests combats time pressures and difficulties to reach end users (Detweiler 2007). The expert then analyses the results and chooses the best solution for each usability problem (**Stage 4**). Meanwhile, if developers are constructing a new MTP during the Second Sprint, the UX expert should specify the usability test(s) before the *sprint review* meeting, so a new iteration of the usability improvement method starts for the new product increment (new **Stage 1**). Note that performing all these stages of our method in one sprint is possible thanks to the set of automated tools that we

have built and the fact that these tools allow the UX expert to work independently of developers. Again, during the *sprint review* meeting, the UX expert chooses some of the participants to run the tests while the product increment is presented. After the *sprint review*, the expert analyzes the test results to discover usability problems (new **Stage 2**).

3) **Third Sprint**: during *sprint planning* of the Third Sprint, the UX expert presents the best solutions discovered in Stage 4 of the previous sprint (that is, the solutions to the usability problems found in the first product increment), so new SBI can be generated for the team to implement the necessary changes at the server side (**Stage 5**). With this, one iteration of our usability improvement method will finish. Also during the *sprint planning*, the UX expert presents the problems found at the end of the previous sprint (Stage 2 of the 2nd iteration of the usability improvement method), for the P.O. to consider as SBI for the expert to test. The sprint continues as the previous one, designing and assembling alternative versions (**Stage 2**), designing tests for these new versions and executing them (**Stage 3**), choosing the best version (**Stage 4**) and specifying the tests for the next product increment (**Stage 1**). The Third Sprint will repeat for every potentially shippable product increment or MTP.

Figure 12 depicts the above description, showing the main events of the agile development cycle, together with the interwoven stages of our usability improvement method. The figure also evidences how our integration proposal allows evaluating the quality of every potentially shippable increment right after development, and allows fixing its problems in the following sprint.

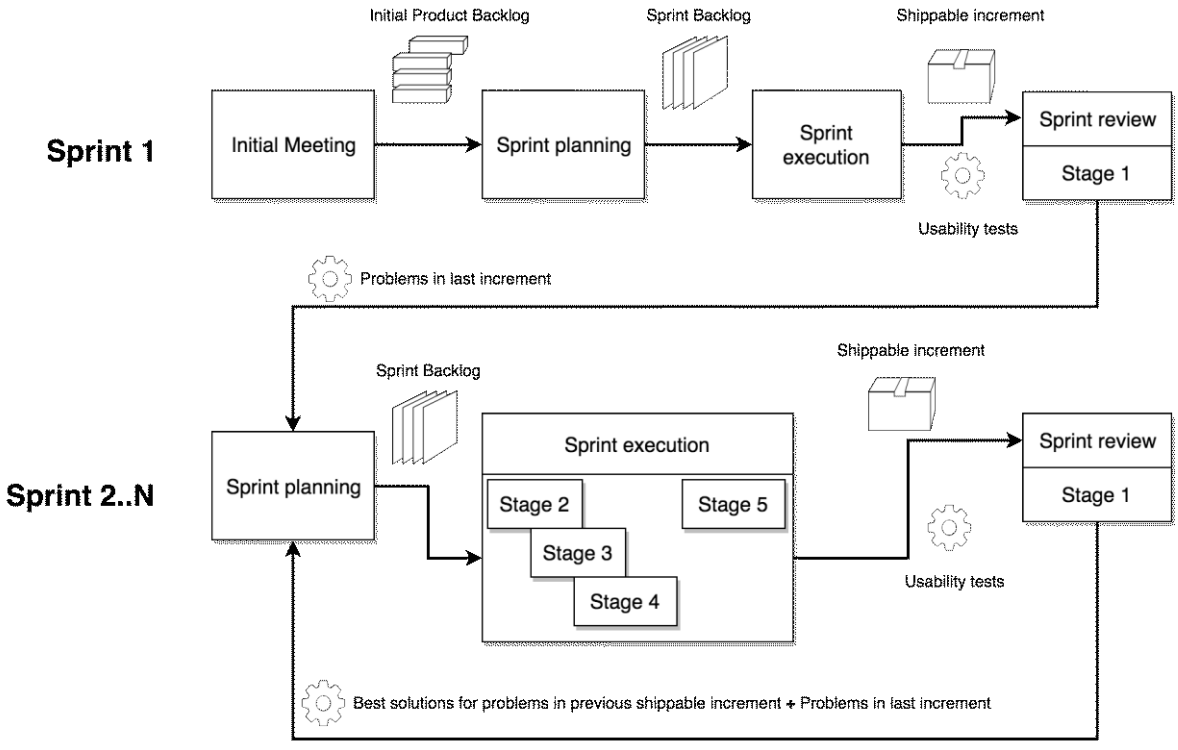


Fig. 12: Integration of agile development events and usability improvement stages

6. CASE STUDIES

We have developed our method of usability improvement and toolkit in light of the experience acquired while studying the benefits of usability and accessibility refactorings and conducting several experiments with them ((Garrido et al. 2013a, 2013b; Grigera et al. 2016)). The first part of this section describes one of these experiments, conducted to assess the improvement achieved with usability refactorings in e-commerce applications (Grigera et al. 2016). This previous work shows the lessons learned, which shed light into the importance of creating a systematic, iterative method to improve usability, and that attends the need to try different solutions since some refactorings do not show improvement in all contexts. The second and third parts of this section present new experiments where we applied our method.

6.1 Case Study 1: E-commerce Applications

The goal of this experiment was to evaluate usability before and after applying refactorings in web applications with complex business processes, particularly e-commerce applications. We used two web applications as *objects* for the experiment: an online store (Z), created with the framework Zencart, and an auction website (W), created with the framework Webid.

Both applications allow users to register, edit their account data, browse through products organized in categories, and acquire products. The means to acquire products differs between both applications: while in the online store users buy products directly through a traditional shopping cart and checkout process, in the auction website they place bids and pay for auctions if they win.

We conducted our experiment with *test subjects* in two locations, Argentina (Ar replication) and Spain (Sp replication). In the Sp replication the volunteers were 22 students of the Degree in Information and Documentation of the University of Valencia (Spain). In the Ar replication we recruited 27 students of the Degree of Computer Science of the University of La Plata (Argentina); 22 of them were undergraduate students, and the remaining 5 were PhD students, most of them working daily in the development of software systems. Each subject worked with both websites in such a way that a particular subject never used the same object twice.

The *use cases* tested in both application were: (1) Register to the website; (2) Update user registration data; and (3) Search for a number of products. Moreover, in application Z we tested (4z) Make a purchase with several products, while in application W we tested two more use cases: (4w) Make a bid on a product and (5w) Sell a product. We refer the reader to our earlier publication (Grigera et al. 2016) for further details of tasks and refactorings involved in each usability test.

What's worth to note is that all use cases listed above were evaluated together, that is, in a single iteration of the usability improvement method presented in this article. Moreover, to measure the usability gain after refactoring, we didn't make refactorings "compete" against each other but only with the non-refactored (NR) versions of each task. This means that we had a single solution for each usability problem found with the original applications. Lastly, not all refactorings were implemented at the client, but 4 out of 21 had to be implemented at the server.

In all cases, our purpose was to measure *usability in use* as we showed in Section 3.2.2. Thus, we defined the following metrics per task: (**M1**) a CategoryObservation for *effectiveness in use,* with two possible values: *1*, which means that the activity was finished successfully; and *0*, which means that the activity failed; (**M2**) a Measurement of completion time in seconds to calculate *efficiency in use*, and (**M3**) a CategoryObservation to measure *satisfaction in use* with questions that take for an answer a value in a 5-point Likert scale. Generally, these likert-scaled statements stated assertions on the activities' ease of use, or general satisfaction. For instance, after Task 3 on the Zencart website (requesting a product search), the assertion that pops up is: "It was easy to search for a specific product in the catalog of the store".

At the time of experiment execution, the *ScPlay* tool was distributed among subjects which found it easy to install. In this case study, the evaluator did not create a leading case. Figure 13 shows a screenshot a *ScPlay* for this experiment.

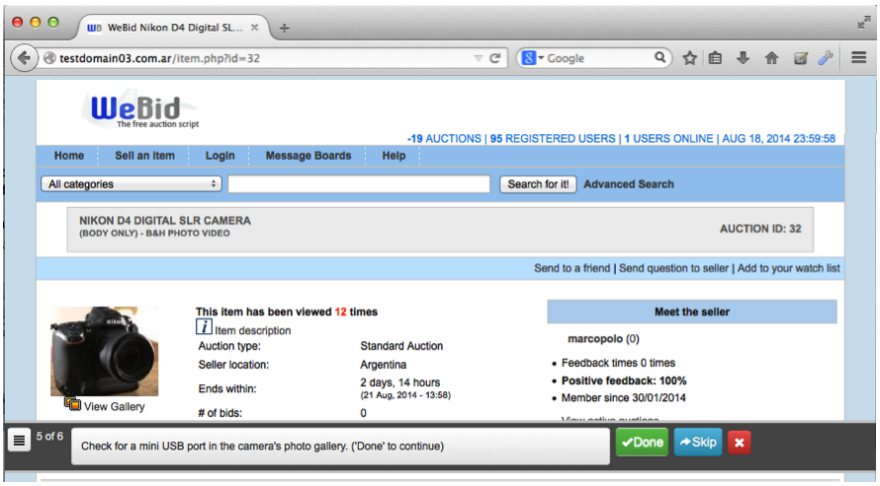


Fig. 13. ScPlay on the WeBid application

At the beginning of the experiment, *ScPlay* displayed a demographic questionnaire to gather the background of the subject, their experience in interacting with web applications, and their knowledge of software development. Next, the *ScPlay* guided subjects through each interaction step, providing extra data when necessary (e.g., a test credit card number), and computing metrics M1, M2 and M3 while doing so.

We learned three insightful lessons with this experiment. The first lesson was that not all refactorings show a usability improvement under all circumstances. In about half of the cases there are other factors to consider like the experience of users and the type of software system. This conclusion showed the importance of trying different, alternative refactorings to solve a particular problem; i.e., it showed the need for A/B testing. The second lesson learned was regarding the drawbacks of testing several use cases in a single test round: subjects get tired or frustrated more easily, creating doubts in the fairness of results, while it is much harder for evaluators to perform the analysis. This lesson made us realize the need for an iterative, systematic method to improve the usability of a single unit of functionality at a time. The third lesson was regarding the usefulness of some of the tools presented in this work to considerably simplify the mechanics of a usability evaluation involving such a number of subjects. Particularly, test subjects found *ScPlay* easy to use and very helpful in guiding them

through the tasks and steps, while it was extremely convenient for the evaluators to have the measurements be computed automatically. The *ScRec and ScEdit* tool resulted highly valuable to configure the tests without needing to type every step. Also, the *CSWR Framework* was used to instantiate most of the refactorings: 17 out of the 21 refactorings, while only 4 had to be implemented at the server side.

6.2 Case Study 2: A Traffic Ticketing Application

This experiment was conducted applying our method to incrementally improve usability of a web application as it was built. The *object* of the experiment was a small web application for managing traffic violation tickets. In this section we show two iterations of the method for two separate use cases: create new tickets for different traffic violations and searching for existing ones. The development team was assembled with 3 advanced students of Computer Science, and 2 of the authors played the role of UX evaluators. The goal was also to try different solutions for the same bad smell thus creating several test branches.

The metrics defined in the first stage of both iterations were similar to the previous experiment: (**M1**) a CategoryObservation for effectiveness (whether the task was successfully completed); (**M2**) a Measurement for efficiency (time in ms) and (**M3**) a CategoryObservation for satisfaction (by a standard SUS Questionnaire (Brooke 1996) scored with a 5 point likert scale). Figure 6 showed earlier the *ScRec* tool on the traffic tickets system ("Multas de Tránsito" in Spanish).

The tests were executed remotely thanks to the *ScPlay* tool (which was depicted in Fig. 8 running a test on this website). Test *subjects* were recruited online, in total 16 subjects with ages ranging from 16 to 45 ($\bar{x}$ 29.75, s 7.8). The subjects' total browsing hours spent per day ranged between 1 and 12 ($\bar{x}$ 5.44, s 3.4). In each iteration of the method the evaluators tested a different use case, and for each one they assembled two alternative versions for solving the existing usability problems. The subjects were evenly distributed, and both iterations counted with 6 users for the original version and 5 for each alternative. As Nielsen asserts, even running a small test with 5 subjects can detect about 85% of the usability problems on a website (Nielsen 2000).

The first usability test consisted in one task: Create a new traffic ticket. Subjects were provided with specific data, namely license plate, violation type, payment due date and address, all appearing in the bottom bar of *ScPlay*, together with the instructions for the current step. The form to create a traffic ticket in the original application provided a plain text field to enter the due date, which users found quite inconvenient while executing the test in Stage 1, since it didn't provide any hint about the date format expected until the submit button was pressed and the form data was validated. In Stage 2, evaluators matched this problem with the smell "Unformatted input" (Grigera et al. 2017), and created 2 versions or alternative solutions applying a different CSWR in each one: *Add Date Picker* and *Format Input* (see Table 6 in Appendix A). The test results of the stages 1 and 3 of the 1st iteration of the usability improvement method are displayed in Table 1. All results are averaged.

The 2nd iteration of the method exercised a new use case: Search for a specific ticket. Subjects had to find out the due date of a traffic ticket for a certain license plate and violation type. Test results from Stage 1 of the 2nd iteration evidenced problems with the violation type field in the search form, since it was a plain text field providing no hints of valid inputs and thus prone to errors. This problem was matched with the bad smell "Free input for limited values" (Grigera et al. 2017) in Stage 2, and 2 versions were created by applying alternative CSWRs: *Turn Text Field into Select* and *Turn Input into Radios* (see Table 6). The results of the test execution in stages 1 and 3 of the 2nd iteration appear in Table 2.

Table 1. Results for Case Study 2: 1st iteration

Version	Effectiveness	Efficiency	Satisfaction
Original	1.0	1m 39s	70.8
Add Date Picker	1.0	54s	61.5
Format Input	1.0	1m 38s	78.0

Table 2. Results for Case Study 2: 2nd iteration

Version	Effectiveness	Efficiency	Satisfaction
Original	0.83	51s	65.4
Text Field into Selects	0.8	44s	56.5
Turn Input into Radios	1.0	46s	74.5

The reader may note in the results that there is no particular solution that improves all metrics, which is the lesson that we learned from this experiment. Kohavi and Longbotham point out that it is frequent to have an experiment improve some metrics while it hurts others, and for this reason they propose to define a single deciding criterion called Overall Evaluation Criterion (OEC) (Kohavi and Longbotham 2015). In our case study, the evaluators defined an OEC composed by all 3 metrics with different weights, where they prioritized satisfaction. In particular, for each refactored version they added the proportional improvement for the 3 metrics over the original version, weighing the terms in such way that satisfaction had the higher significance. Using this OEC, in Stage 4 of the 1st iteration, the winning refactoring was *Format Input*. In the 2nd iteration, the winning refactoring was *Turn Input into Radios*.

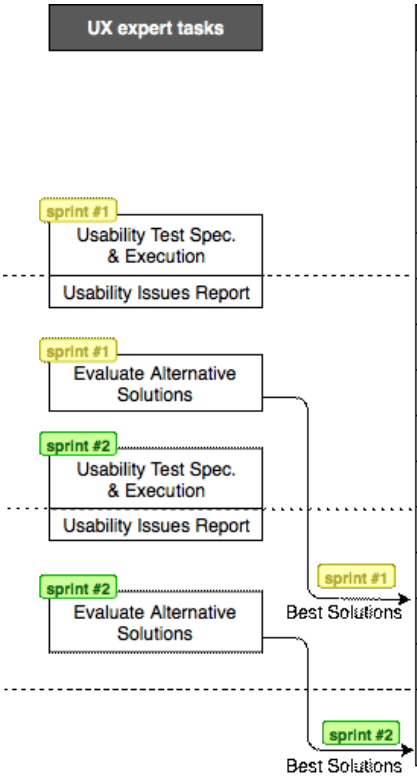
In this case study, the tools were a great assistance, providing an inexpensive but sound analysis and improvement mechanism of the usability of a recently implemented feature, right on the next sprint it was developed. This experience reveals that usability analysis and correction can be incorporated into a standard agile process without imposing an excessive additional burden to developers, and making it simpler and incremental for the UX staff.

6.3 Case Study 3: An Online Shoe Store

Following the same design of the experiment described in Section 6.2, we ran another experiment with an object from a different domain: an e-commerce website for shopping shoes. In this case, the application is larger, since it involves complex business processes of

online shops, and the experiment exercises tasks that are very representative of e-commerce applications: search and check out, also experimenting many of the most usual usability problems that can be found on these tasks. During this experiment, we also logged the development process in detail, and added more refactorings per task.

In order to build the shoe store application, the development team followed a classical Scrum process, with a 15-day sprint period. The development team consisted in a tech lead and 2 developers. The initial Product Backlog contained 12 stories (the first 12 issues in Figure 14). For estimation, Story Points were used as an abstract metric to measure the effort that every issue represents without spending an extensive amount of time estimating it in detail. During *sprint planning*, the stories were divided in 3 sprints. Each sprint included the development of a user interface related to a use case and the output in each case was a Minimum Testable Product (MTP) to be demonstrated during the *sprint review*. The application was developed as planned, except from minor adjustments that needed to be made on the checkout process (particularly, SS-6).



Issue ID	Description	Story Points	Sprint
SS-1	Login/Logout and auth	6	1
SS-2	Registration	3	
SS-3	Search for a product	2	
SS-4	Browser product details	2	
SS-5	Checkout: Confirm	2	
SS-6	Checkout: And CC information	2	2
SS-7	Checkout: Add delivery info	5	
SS-8	View shopping cart	3	
SS-9	Add product to shopping cart	2	
SS-10	Add product to shopping cart	3	
SS-11	Change account info	5	3
SS-12	Leave feedback with the contact form	3	
SS-13	Change Widget: Submit button for magnifying glass	4	
SS-14	Change Font: US/EU scale conversion	1	
...	...	...	4

Fig. 14. Shoe Store Product Backlog and Sprint Planning

Regarding the usability improvement method that run in parallel with development (left side of Figure 14), the team had one UX expert that applied it. The test *subjects* sample used to execute the user tests consisted of a total of 16 subjects with ages ranging from 14 to 48 ($\bar{x}$ 31.5, s 19.92). The total browsing hours spent per day ranged between 1 and 12 ($\bar{x}$ 6.06 s 3.19).

During the development there were 2 new refactorings needed by the expert for solving the usability smells, which were not present in the *CSWR Repository*. For those, during a *daily meeting* the expert requested one developer to create them, which was possible because their implementation did not take more than 2 hours each.

During the *execution* of the **first sprint**, the expert designed a usability test related with the use case to search for a product and browse its details, involving stories SS-3 and SS-4. The test consisted of 3 tasks: 1) Search for "boots" using the website search tool, 2) find a specific product and 3) select a specific shoe size in EU scale (the website showed the US sizes, so subjects had to make the conversion). The test was run at the end of the first sprint, during a *sprint review* meeting, using the tool *ScPlay*. The original application featured a very simple search box with a placeholder text that hinted to "search for products". The problems highlighted by the test of this story (SS-3) were that the search box was not a very noticeable widget, and that the results showed only after hitting "Enter". In the case of task 3 (SS-4), subjects had problems to find the conversion from US to EU scale.

During the *sprint planning* of the **second sprint**, backlog items were generated for the expert to find alternatives to solve the usability problems found (left-hand side of Fig. 14), together with new user stories for the team to develop (right side of Fig. 14). The expert first related each of the problems found with a usability smell. For the noticeability problem, the matching smell was "Undescriptive Element" and 2 CSWR were selected as alternative solutions for it: *"Change Widget"*, replacing the submit button for a magnifying glass, and the alternative *"Change Font"* which was a new refactoring that simply alters the aspect of a given text to make it, in this case, more prominent. For the results appearing only after the search request, with the consequence of some empty results, the expert matched it with the smell "Scarce search results", and applied also 2 alternative CSWRs: *"Add Autocomplete"*, in order to improve the shopper's confidence in the search, and *"Live Results"*, also a new refactoring that displays and filters the results in the client side, as the customer writes the query. For task 3, the matching smell was "Undescriptive Element", and the expert applied 2 CSWRs to allow the subjects to find the shoes sizes' conversion table. One refactoring was *"Add Tooltip"*, which added a tooltip to the legend that states "Shoe Sizes are in US scale" with the conversion table. The alternative refactoring was *"Turn Attribute into Link"*, which converted the same legend into an anchor leading to a page with the conversion table. The test was executed remotely by the 16 recruited subjects during a period of 1 week, which left the expert another week to take care of designing the usability test for the stories that were being developed in the second sprint. The test results of the first iteration of our usability improvement method are displayed in Table 3. All results are averaged. After the test, the best refactoring was selected for each smell by using the same Overall Evaluation Criterion (OEC) from the previous experiment. New items were latter added to the Product Backlog to implement the best refactorings at the server (stage 5 of the usability improvement method) in the third sprint. These new backlog items can be seen in Figure 14 as SS-13 and SS-14.

Table 3. Results for Case Study 3: 1st iteration

Version	Effectiveness	Efficiency	Satisfaction
Original	80,00 %	0m 59s	60,50
Change Widget	57,14 %	0m 48s	**81,79**
Change Font	**75,00 %**	**0m 47s**	70,31
Add Autocomplete	**71,43 %**	0m 50s	66,07
Live Results	62,50 %	**0m 46s**	**84,06**
Add Tooltip	**66,67 %**	0m 50s	**82,78**
Turn Attribute into Link	**66,67 %**	**0m 45s**	65,00

Going back to the *sprint review* at the end of the **second sprint**, the expert was able to run the usability test for the increment being deployed, related to the use case for checking out products. This time, there were two tasks: subjects had to add 2 of a same product to the shopping cart, and then complete the checkout process. For the first task, the original version included a simple text input for entering the product's quantity, which was prone to error. For the checkout process, the original version implemented validation only after the request, i.e., on the server side. This caused failed submissions because it wasn't clear which fields were mandatory (some did not fill out postal code, some did not fill out telephone, both mandatory).

During the *sprint planning* of the **third sprint**, backlog items were generated for the expert to solve the usability problems found, together with new user stories for the team to develop, this time including those that solved the UX issues detected in the first sprint (see Figure 14). During this sprint the expert related the problem of the simple text input prone to errors with the usability smell "Free input for limited values", and applied 2 refactorings to create 2 alternative versions: *"Text Input into Select"* and *"Change Widget"* (converting the text input into a label with "+" and "-" buttons to increment and decrement the amount). For the checkout task, the problem was matched with the smell "Late Validation", and the 2 alternatives to improve this interaction where *"Inline Validation"*, which validated the mandatory fields right after moving the cursor away (blur event), and *"Client-side Validation"* which validates all fields at once, but preventing the submission, that is, on the client side. The test was executed remotely by the 16 subjects. Again, after the test, the best refactoring for each case was selected using the OEC previously presented. Results are displayed in Table 4. In this case, the winning refactoring for the "Free input for limited values" smell was *"Text Input into Select"*. Even if the satisfaction metric was slightly under the competing refactoring *("Change Widget"*), the perfect effectiveness and considerably better efficiency (19 seconds less in average) were decisive in the final score. The same happened for the selection of the winning refactoring to the "Late Validation" usability smell.

The winning refactoring, *"Client-side Validation"*, was also slightly worse in satisfaction (less than 1 unit), but substantially better in both effectiveness and efficiency (see Table 4).

Table 4. Results for Case Study 3: 2nd iteration

Version	Effectiveness	Efficiency	Satisfaction
Original	80,00 %	0m 54s	73,50
Text Input into Select	**100,00 %**	**1m** 1s	80,00
Change Widget	87,50 %	1m 20s	**80,63**
Inline Validation	87,50 %	1m 24s	**80,94**
Client-side Validation	**100,00 %**	**0m 57s**	79,64

7. CONCLUSION AND FUTURE WORK

Even when existing literature concludes that usability is an essential aspect related to the quality of web applications, its evaluation is hardly undertaken. The main reason is that performing usability evaluations is expensive, and consequently, it is hard for most companies to keep it in synch with development. At the current rate of innovation, being out of synch implies losing business. From our point of view, there exist both a methodological and a technical gap that make it very difficult to accomplish usability evaluation each time a new user interface is built, especially when agile methodologies are used. Current approaches show an unhealthy interdependence between the development team and the usability expert. Under this interdependence, in order to improve the quality of the latest product increment, while developers depend on the expert to find usability problems and design solutions, the expert depends on developers to get concrete implementations of these solutions to test them and decide which one is the best.

In this paper, we propose a method for improving usability based on A/B testing and refactoring. Furthermore, we support this method with a set of tools that empowers usability experts to design and conduct user tests remotely, gather metrics automatically, analyze them, and assembly alternative solutions to try out different ideas without compromising current development. In the last stage, developers just intervene to implement the optimal solution. Thus, our method and toolkit makes it affordable to learn from user feedback as early as there is a Minimum Testable Product (MTP). By untangling the unhealthy interdependence, we allow developers and usability experts to work in parallel, which is necessary if usability evaluation aims to be done sustainably in agile methodologies. We still support multidisciplinary teams and pair programming between developers and UX experts, but only when it creates mutual benefit, for instance when defining acceptance tests.

We have designed and implemented specific tools for the different steps of our method. The core of the approach is the CSWR framework, which makes possible to refactor web user interfaces on the client-side, without depending on editing the application source code, neither requiring programming skills. We have also proposed an approach to integrate our usability improvement method with an agile development process. This integration is compatible with the evidence that we collected from local development companies about

usability practices. Last but not least, we have also conducted experiments that demonstrate the feasibility of the method for each MTP constructed and the usefulness of the tools.

Our future work includes further experimentation about the adoption of our method and tools by UX experts. We also plan to expand the CSWR Repository and improve the tools with the feedback learned from UX experts. For example, we envision extending the Test Analysis tool with new visualizations of results, the CSWR Visual Instantiation tool to create new refactorings easily. Moreover, we plan to integrate other tools we have built, that allow for the automatic detection of usability smells (USF), as well as the automatic instantiation of refactorings. Since USF does not require the specification of tasks for the users to follow, it provides a different perspective to the UX expert, aligned with the current practice in A/B testing tools. Finally, we would like to refine our integration approach with the specifics of other agile methods, like Extreme Programming and Lean Software Development.

REFERENCES

Benigni, G., Gervasi, O., Passeri, F., and Kim, T. (2010) USABAGILE_Web: a web agile usability approach for web site design. ICCSA (2) - Lecture Notes in Computer Science (LNCS), 6017, 422–431.

Brooke, J. (1996) SUS - A quick and dirty usability scale. Usability evaluation in industry, 189, 4–7.

Burzacca, P., and Paternò, F. (2013) Remote Usability Evaluation of Mobile Web Applications. In Proceedings of the 15th Int. Conf. on Human-Computer Interaction Vol. 1, pp. 241–248.

Carta, T., Paternò, F., and Santana, V. De (2011) Web usability probe: a tool for supporting remote usability evaluation of web sites. In INTERACT 2011. LNCS 6949 pp. 349–357.

Da Silva, T.S., Silveira, M.S., De O. Melo, C., and Parzianello, L.C. (2013) Understanding the UX designer's role within agile teams. In LNCS Vol. 8012, pp. 599–609.

Detweiler, M. (2007) Managing UCD within agile projects. Interactions, 14, 40.

Diaz, O., and Arellano, C. (2015) The augmented web: rationales, opportunities, and challenges on browser-side transcoding. ACM Transaction on the Web, 9, 8:1-8:30.

Distante, D., Garrido, A., Camelier-Carvajal, J., Giandini, R., and Rossi, G. (2014) Business processes refactoring to improve usability in E-commerce applications. Electronic Commerce Research, 14, 497–529.

Düchting, M., Zimmermann, D., and Nebe, K. (2007) Incorporating user centered requirement engineering into agile software development. In 12th Int. Conf. Human-computer interaction pp. 58–67.

Fernandez, A., Insfran, E., and Abrahão, S. (2011) Usability evaluation methods for the web: A systematic mapping study. Information and Software Technology, 53, 789–817.

Fernandez, A., Abrahão, S., and Insfran, E. (2013) Empirical validation of a usability inspection method for model-driven Web development. Journal of Systems and Software, 86, 161–186.

Firmenich, D., Firmenich, S., Rivero, J.M., Antonelli, L., and Rossi, G. (2016) CrowdMock: an approach for defining and evolving web augmentation requirements. Requirements Engineering, 1–29.

Fowler, M. (1997) Analysis Patterns: Reusable Object Models. Addison Wesley. Addison Wesley.

Fowler, M. (1999) Refactoring: improving the design of existing code, 431 p. Addison-Wesley.

Garrido, A., Rossi, G., and Distante, D. (2011) Refactoring for usability in web applications. IEEE Software, 28, 60–67.

Garrido, A., Rossi, G., Medina, N.M., Grigera, J., and Firmenich, S. (2013a) Improving accessibility of Web interfaces: refactoring to the rescue. Universal Access in the Information Society, pp. 1–13.

Garrido, A., Firmenich, S., Rossi, G., Grigera, J., Medina-Medina, N., and Harari, I. (2013b) Personalized web accessibility using client-side refactoring. IEEE Internet Computing, 17, 58–66.

Garrido, A., Firmenich, S., Rossi, G., Grigera, J., Medina, N., and Harari, I. (2013c) Personalized web accessibility using client-side refactoring. IEEE Internet Comp, 17, 58–66.

Genov, and Alex (2005) Iterative usability testing as continuous feedback: a control systems perspective. Journal of Usability Studies, 1, 18–27.

George, C.A. (2005) Usability testing and design of a library website: an iterative approach. OCLC Systems & Services: International digital library perspectives, 21, 167–180.

Grigera, J., Garrido, A., Panach, J.I., Distante, D., and Rossi, G. (2016) Assessing refactorings for usability in e-commerce applications. Empirical Software Engineering, 21, 1224–1271.

Grigera, J., Garrido, A., Rivero, J.M., and Rossi, G. (2017) Automatic detection of usability smells in web applications. International Journal of Human-Computer Studies, 97, 129–148.

Hartson, H.R., and Castillo, J.C. (1998) Remote evaluation for post-deployment usability improvement. In Proceedings of the working conference on Advanced visual interfaces - AVI '98 pp. 22–29. ACM Press, New York, New York, USA.

Hartson, H.R., Andre, T.S., and Williges, R.C. (2003) Criteria For Evaluating Usability Evaluation Methods. International Journal of Human-Computer Interaction, 15, 145–181.

Hassenzahl, M. (2006) Hedonic, emotional, and experiential perspectives on product quality. Encyclopedia of human computer interaction, 266–268.

ISO, I. (2011) ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.

Jurca, G., Hellmann, T.D., and Maurer, F. (2014) Integrating agile and user-centered design: A systematic mapping and review of evaluation and validation studies of agile-UX. In Proceedings - 2014 Agile Conference, AGILE 2014.

Kohavi, R., and Longbotham, R. (2015) Online Controlled Experiments and A/B Tests Motivation and Background. Encyclopedia of Machine Learning and Data Mining.

Kohavi, R., Longbotham, R., Sommerfield, D., and Henne, R.M. (2009) Controlled experiments on the web: survey and practical guide. Data Mining and Knowledge Discovery, 18, 140–181.

Lee, J.C., and McCrickard, D.S. (2007) Towards Extreme(ly) Usable Software: Exploring Tensions Between Usability and Agile Software Development. In Agile 2007 pp. 59–71.

Nielsen, J. (2000) Why You Only Need to Test with 5 Users. https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/. Last accessed June 16, 2018.

Nielsen, J., and Loranger, H. (2006) Prioritizing Web Usability, 406 p. (C. Peri, Ed.). Pearson Education.

Obendorf, H., and Finck, M. (2008) Scenario-based usability engineering techniques in agile development processes. Proceeding of the twenty-sixth annual CHI conference extended abstracts on Human factors in computing systems - CHI '08, 2159.

Paganelli, L., and Paternò, F. (2003) Tools for remote usability evaluation of Web applications through browser logs and task models. Behavior research methods, instruments, & computers : a journal of the Psychonomic Society, Inc, 35, 369–378.

Panach, J.I., Juristo, N., Valverde, F., and Pastor, O. (2015) A framework to identify primitives that represent usability within Model-Driven Development methods. Information and Software Technology, 58, 338–354.

Rubin, J., and Chisnell, D. (2008) Handbook of Usability Testing: Howto Plan, Design, and Conduct Effective Tests. Wiley.

Rubin, K. (2012) Essential Scrum: A Practical Guide to the Most Popular Agile Process. Addison-Wesley.

Salvador, C., Nakasone, A., and Pow-Sang, J.A. (2014) A systematic review of usability techniques in agile methodologies. In Proceedings of the 7th Euro American Conference on Telematics and Information Systems - EATIS '14.

Santana, V.F. de, and Baranauskas, M.C.C. (2015) WELFIT: A remote evaluation tool for identifying Web usage patterns through client-side logging. International Journal of Human-Computer Studies, 76, 40–49.

Schissel, J. (2014) From MVP to MTP: Accelerating Development at Your Company. https://www.linkedin.com/pulse/20140710165313-4729218-from-mvp-to-mtp-accelerating-development-at-your-company/. Last accessed June 16, 2018.

Silva da Silva, T., Martin, A., Maurer, F., and Silveira, M. (2011) User-Centered Design and Agile Methods: A Systematic Review. Agile Conference (AGILE), 2011, 77–86.

Speicher, M., Both, A., and Gaedke, M. (2014) Ensuring Web Interface Quality through Usability-Based Split Testing. In Icwe, LNCS 8541 pp. 93–110.

UID (2018) AttrakDiff. User Interface Design GmbH. http://attrakdiff.de/. Last accessed June 16, 2018.

Urbieta, M., Firmenich, S., Maglione, P., Rossi, G., and Olivero, M.A. (2017) A Model-driven Approach for Empowering Advance Web Augmentation - From Client-side to Server-side Support. In WEBIST pp. 444–454.

Williams, L., and Cockburn, A. (2003) Agile software development: It's about feedback and change. Computer.

Yoder, J.W., Balaguer, F., and Johnson, R. (2002) From analysis to design of the observation pattern.

APPENDIX A

Table 6 below shows the listing of the existing CSWRs that are available in the *CSWR Repository* (currently 31). The table shows in the 1st column, the refactoring name, in the 2nd column, the usability smell(s) that the refactoring may solve, and the 3rd column describes the change that each refactoring produces on webpage elements. We tried to group refactorings that may solve the same or a similar smell.

The reader may notice that some of the refactorings apply changes that specially improve accessibility, which may also be a goal of the UX expert.

Table 6. Existing CSWRs in the Repository

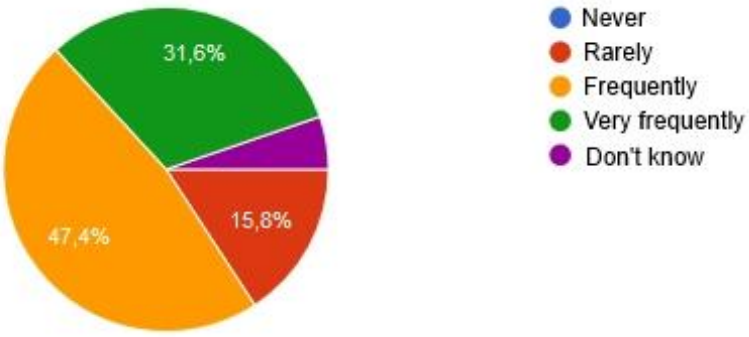
#	Refactoring	Usability Smells	Change applied
1	*Split page*	Saturated page; Large page	Divide a complex page into a structure of interrelated pages, reorganizing semantic, layout, and navigation structure so that each new page is simpler and more cohesive.
2	*Remove unnecessary content*	Obsolete content; Duplicated content; Large page	Remove contents that confuse rather than help users to complete a task
3	*Remove Duplicate Process Link*	Duplicated content	Remove redundant links to simplify process pages.
4	*Merge pages*	Long navigation paths	Redistribute contents to bring together pieces of information that belong in the same page (content-wise) for convenience.
5	*Focus-dispersed content and/or operations*	Confusing organization; Long navigation paths	Redistribute contents and/or operations that share a process, bringing them together.
6	*Fix menu*	Excessive backward scrolling toward menu	Leave a menu fixed in the page so there is no need to scroll up to find it.
7	*Replace non-accessible menus by list of links*	Disguised navigation structure	Accessibility refactoring. Aims at laying out options that otherwise are hidden in pull down menus. Easier on screen-readers.
8	*Show all structural links at once*	Disguised/spread navigation structure	Bring together navigation options to make easier to find.
9	*Add size indicators*	Unpredictable size; Users quitting before completing a goal	Add indicators for long list, tables or steps in a process to let the user know their extent.
10	*Rename Link*	Misleading link	Change the test of a link to describe it more precisely
11	*Replace Image for Text*	Missing Alt text for images	Accessibility refactoring for screen readers. Describe an image textually, for easier recognition of contents.
12	*Postpone selection*	Reading twice through an element to select it	Accessibility refactoring for screen readers. Change the selection checkbox placement to the end so users can read the content before deciding whether they want to check it for a bulk action.
13	*Distribute Menu*	Unnecessary bulk action; Inaccessible bulk action	In a list of items with checkboxes for bulk actions, add the most common actions as individual links next to each item, so it is easier to apply on a single one.

14	*Add Processing Page*	No processing page	Add a “loading” overlay to make clear that the web application is processing as opposite to a frozen UI.
15	*Add Autocomplete*	Free input for limited values; Scarce search results	Add assistance by autocomplete features, so users need to type less and be reassured of the options.
16	*Turn Input into Radios*	Free input for limited values	Turn a text field with a narrow set of choices into a radio buttons set, adding an “other” option to leave the choice of entering an unexpected option, but making it easier for the popular choices.
17	*Turn Text Field into Select*	Free input for limited values	Turn a text field with a strictly closed set of options into a select box with those options.
18	*Add Date Picker*	Unformatted input	Add a date picker widget to a text field intended to capture dates, avoiding format errors at typing.
19	*Date Input into Selects*	Unformatted input	Turn a text field intended to capture dates into a set of 3 select boxes for day – month – year
20	*Format input*	Unformatted input	Provide inline formatting for an input field that only accepts data of a fixed format, like credit cards, telephone numbers or dates.
21	*Provide Default Option*	Wrong Default Option	Alter or set the default option in a select box or radio buttons set to be the most popular one.
22	*Client-side Validation*	Abandoned form Late validation No client validation	Add validation on submit to a form that validates at the server.
23	*Inline Validation*	Abandoned form Late validation	Add validation to the mandatory fields in a form right after moving the cursor away (blur event)
24	*Change Widget*	Undescriptive Element; Unresponsive element; Free input for limited values	Replace a UI control by a different one(s), more suitable for the value to be entered
25	*Change Font*	Undescriptive Element	Alter the aspect of a given text
26	*Add Tooltip*	Undescriptive Element	Add tooltip on hover event
27	*Turn Attribute into Link*	Undescriptive Element; Unresponsive element	Turn a static element, such as a text or an image, into a link that navigates away or adds contextual content (i .e. enlarge an image)
28	*Live Results*	Scarce search results	Display the results of a query as the customer types the search text
29	*Resize Input*	Unmatched size input	Set a text input size to the expected values to be entered, hinting the user to what´s expected.
30	*Link to Top*	Overlooked content	Add a link or button that, when clicked, scrolls to the top of the page, preventing users to manually scroll a very long way up.
31	*Add Link*	Distant content; Long navigation paths	Provide a new link as a shortcut to navigate an otherwise long path of nodes

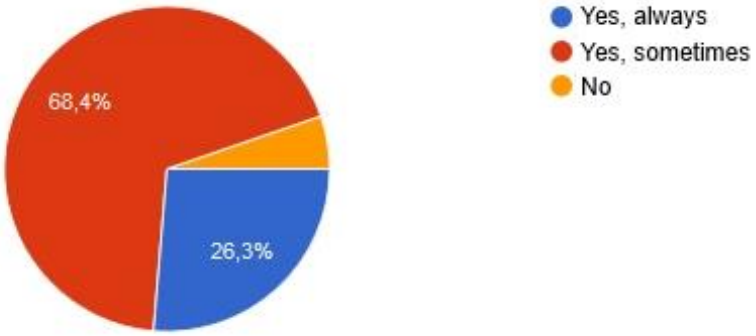
APPENDIX B

This appendix provides details of a survey about the adoption of usability practices that we conducted among some software development companies in the area of La Plata and Buenos Aires, Argentina. There was a total of 19 developers that answered the survey, composed of 7 questions that we list below, together with the results.

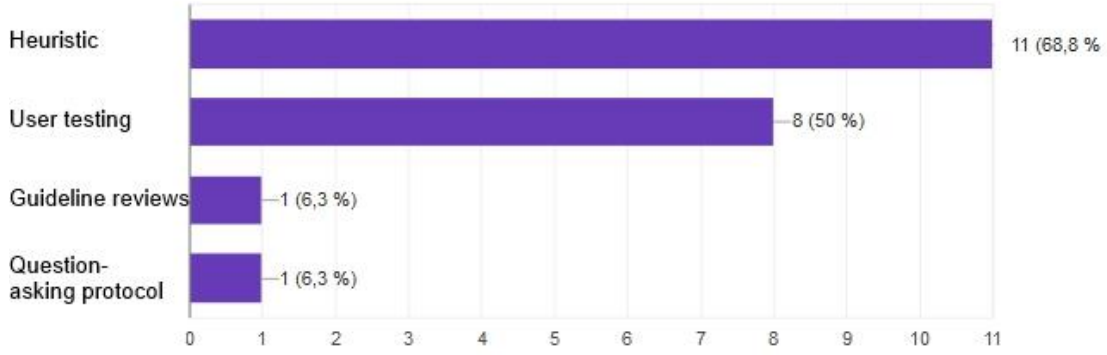
1) In your team, how frequent is that a user interface changes more than once during development or production?



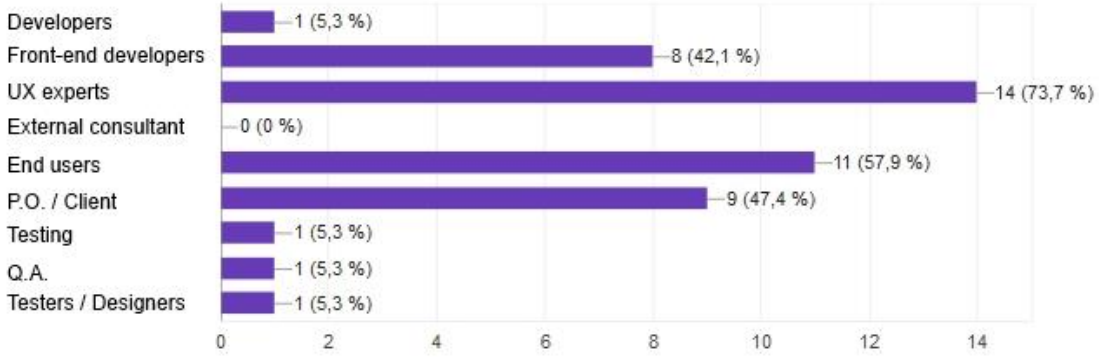
2) Do your projects receive any assessment on usability?



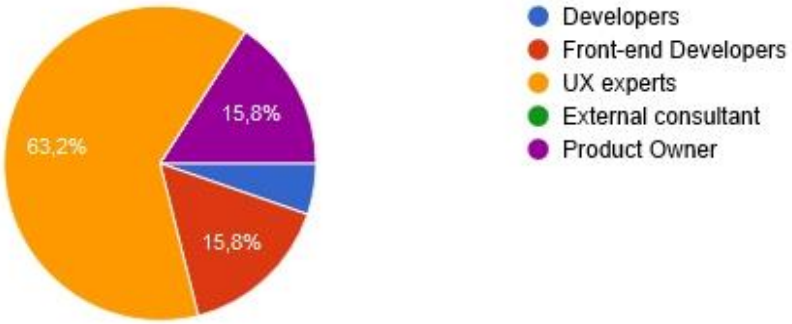
3) If the answer to the previous question was “yes”, what kind of usability assessment is it performed?



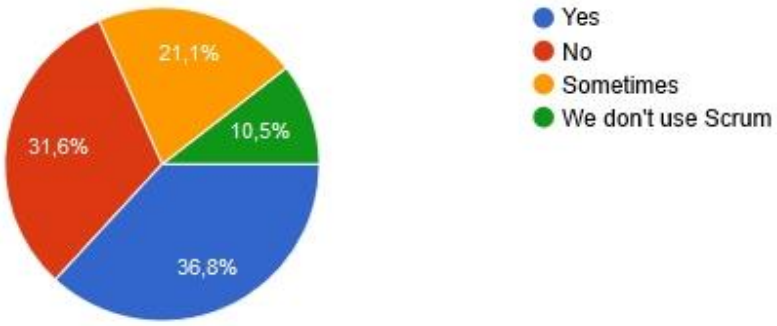
4) Who reports usability problems in your projects?



5) Who proposes and validates solutions to usability problems in your projects?



6) If your team uses Scrum, does your team do Sprint Reviews?



7) Do you think that *users tests* can be conducted during Sprint Review meetings?