Automatic Detection of Usability Smells in Web Applications

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

[1]LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
{julian.grigera, garrido, mrivero, gustavo}@lifia.info.unlp.edu.ar
[2]Also at CONICET, Argentina

Abstract

Usability assessment of web applications continues to be an expensive and often neglected practice. While large companies are able to spare resources for studying and improving usability in their products, smaller businesses often divert theirs in other aspects. To help these cases, researches have devised automatic approaches for user interaction analysis, and there are commercial services that offer automated usability statistics at relatively low fees. However, most existing approaches still fall short in specifying the usability problems concretely enough to identify and suggest solutions. In this work we describe usability smells of user interaction, i.e., hints of usability problems on running web applications, and the process in which they can be identified by analyzing user interaction events. We also describe USF, the tool that implements the process in a fully automated way with minimum setup effort. USF analyses user interaction events on-the-fly, discovers usability smells and reports them together with a concrete solution in terms of a usability refactoring, providing usability advice for deployed web applications.

Keywords: Usability testing, Web-based interaction, Refactoring, Log analysis

1 Introduction

Web applications help us in many of our daily life activities, like shopping, news reading, social interaction, home banking, trip planning or requesting a doctor's appointment. Every day new websites appear broadening our possibilities to accomplish tasks comfortably from home, and yet many times they suffer from usability problems that make them awkward and hard to use. As Nielsen said, *usability rules the web*, and it is crucial for a website's success (Nielsen & Loranger 2006; Gregg & Walczak 2010). Companies acknowledge that competition is so high that they will hardly survive if they do not invest in usability, although it still remains expensive and therefore neglected despite the progress in research and tools (Nielsen & Loranger 2006).

One of the most popular ways of evaluating usability is by conducting usability tests, particularly, user tests (Rubin & Chisnell 2008). The benefit of user testing over inspection methods like heuristic evaluations is that it captures real usage data and users' experiences. The down-side, however, is that it requires recruiting users and spending time and resources for experts first to design the tests and afterwards to analyze the results, discover the problems and find solutions for those problems.

To overcome the need of having the expert manually collecting and comparing test results, different automated approaches exist for remote user testing. Several approaches log user interaction (UI) events and perform some log analysis to help the expert discover usage patterns (Santana & Baranauskas 2015). The results are usually presented with sophisticated visualization tools that allow comparing user event sequences with an optimal sequence. However, these tools rarely provide suggestions to help designers improve their artifacts; the

expert is still needed to detect concrete usability problems in the deviations among event sequences, and find a solution (Fernandez et al. 2011). Moreover, the set of usability problems that can be recognized by comparing event sequences is limited (for instance, a frequently performed activity could be unnecessarily long for all users). In turn, there are many sources of usability guidelines and good practices in the literature, though it is still hard for a developer to identify which of these guidelines address a particular problem that appears on a running application.

Our proposal for overcoming these two issues related to dynamic usability assessment and repair is to extend event logging analysis to report concrete problems that are solvable through refactoring. Being an agile practice, refactoring allows improving usability in an incremental way using feedback from users, even (especially) in already deployed applications (Garrido et al. 2011). Moreover, refactorings are beneficial not only as catalogued, mechanized solutions, but also because each solution is linked to the particular problem or "smell" that it solves. In the case of usability, we call them **usability smells** (Garrido et al. 2011).

In our current work, we aim at providing automatic advice about usability smells of user interaction for deployed web applications. Our automated strategy to usability smell recognition is based on the analysis of user interaction (UI) events. Thus, we extend previous work in this area by linking specific UI events to usability smells, defining new usability smells, and reporting usability smells on-the-fly at an abstraction level which makes it possible to suggest concrete solutions for them in terms of refactorings.

We have implemented the approach in a tool called **USF** (**Usability Smells Finder**). The tool can be used as a service (SaaS) with minimal setup effort, and is able to provide up-to-the-minute advice for deployed web applications. It is implemented in a way that allows for the extension of usability smells' detection strategies. Therefore, it is targeted to a broad audience of practitioners with different levels of usability expertise. On the one hand, usability experts may use USF to get rapid feedback of real interactions from a mass of users, configuring the tool to their needs. On the other hand, developers without usability expertise may use USF after a simple installation, let the tool gather evidence to diagnose usability problems and implement the solutions that it suggests.

The structure of the rest of the article is as follows: the next section presents background on usability refactoring and our catalog of usability smells for user interaction. Section 2 provides background and describes the differences and similarities with related work. Section 3 describes the usability smells used in this work, while Section 4 describes in detail the process of usability smell recognition and reporting, starting from the analysis of UI events, their abstraction to a mid-level concept called usability events, and the filtering and aggregation of usability events into usability smells. Section 5 provides the architecture and implementation details of the tool. Section 6 presents two experiments that we ran to assess the accuracy and versatility of the process and tool, and finally Section 7 concludes the article with contributions and future work.

2 Related Work

In this section we will review different approaches related to different aspects of our proposal. We first provide a background on refactoring and bad smells, their specific application in the context of web usability, and our previous work on the definition of usability smells. We then

review usability evaluation methods in general, and place our work into their classifications. Then, we review log analysis and visualization methods, following with remote user testing and finally we describe the most relevant approaches we found to be closer to our proposal.

2.1 Background on Usability and Refactoring-based Tools

Our work in usability takes many ideas from code refactoring. Refactoring was originally defined by Opdyke as a transformation that preserves behavior, aimed at improving the internal design of code (Opdyke 1992). Later on, Fowler popularized the technique by publishing a catalog of refactorings for object-oriented code (Fowler 1999). The refactorings in Fowler's catalog seek to improve internal quality measures like understandability, extensibility and maintainability of the different components in an object-oriented program like methods, classes and hierarchies, as well as data and conditional expressions. For example, "Extract Method" turns a piece of code into its own method, with an appropriate name that explains its purpose. The power of refactoring lies in helping non-experts to identify potential problems in the target aspect and traverse through a series of small steps towards a good solution for those problems. In the refactoring jargon, these potential problems are called *"bad smells"*, and their presence likely means that the code needs refactoring. For example, the refactoring "Extract Method" is intended to solve smells like "Long Method" and "Duplicate Code" (Fowler 1999).

The refactoring technique became an essential practice of agile methodologies, and its scope was soon extended to other programming paradigms and beyond improving internal qualities of code into improving external qualities of software, like database safety (Ambler & Sadalage 2006), parallel programs' performance (Dig 2011), and web application's navigability (Cabot & Gómez 2008). In 2007, we started working on the application of refactoring to improve the usability of web applications (Garrido et al. 2007). We defined **usability refactorings** as changes to the navigation, presentation or business processes of a web application with the purpose of improving its usability, while preserving the expected functionality and result (Garrido et al. 2011). With these refactorings, developers may attain usability enhancements like a balanced distribution of content in the screen and among pages, a better navigation structure and process workflow, proper support for the user while executing a business process, etc. An example of a usability refactoring is *"Provide Breadcrumbs"*, to help users keep track of their navigation path up to the current page (Garrido et al. 2011).

Similarly to bad smells in code ("code smells" for short), we have defined **usability smells** as indicators of possible problems that need refactoring (Garrido et al. 2011), where the problems relate to any aspect of the quality in use of a web application: effectiveness in use, efficiency in use, or satisfaction in use (ISO 2011). In our earlier work, usability smells were catalogued to be manually recognized at different model levels (Garrido et al. 2011). Examples of these smells are "Absence of meaningful links" (in the navigation model), "Cluttered interface" (in the presentation model), and "Long activity" (in the process model).

In a later work, we developed a framework that allows applying usability refactorings on the client side, thus reducing the cost of changing a running system at the server (Garrido et al. 2013). We also developed a catalog with new usability refactorings and usability smells (Distante et al. 2014). The refactorings in this catalog may be applied either at the model level, in a model-driven development approach or at the client-side, and usability smells may be

discovered in the models or by manually inspecting the results of user tests. An example of usability refactoring from that catalogue is *Change the widget used to execute an activity*, aimed at replacing a widget that has been found awkward to use or produce errors for a more appropriate one, e.g. replacing free textboxes with calendar widgets for selecting dates, or selection boxes for ranged values. In the catalogue, the usability smell that triggers this refactoring is *Risk of error*. Other catalogued usability smells are *Difficult access to information, User confusion*, and *Process inflexibility* (Distante et al. 2014).

Moreover, we have conducted a statistical test to measure the real gain that usability refactorings produce on effectiveness in use, efficiency in use and satisfaction in use (Grigera et al. 2016). For the experiment, detecting usability smells was not a simple task, as it required manual expert intervention to go through the results of user tests.

Finally, in the context of refactoring, our process for smells detection could be compared to the work of Lanza and Marinescu to systematically detect bad smells in code: they use well-known object-oriented metrics and metric-based patterns as detection strategies to identify potential bad smells and structural design problems and provide the appropriate refactorings as recovery means (Lanza & Marinescu 2006). In a similar way, we use UI event patterns to characterize usability smells and suggest usability refactorings for most cases.

2.2 Usability Evaluation Methods

Like Hornbæk said in his review back in 2006 (Hornbæk 2006), usability cannot be directly measured, so researchers must select usability aspects that can be measured and represent valid indicators of usability, like in the model proposed by Seffah et al. (Seffah et al. 2006). There are different ways to classify usability measures and usability evaluation methods, although there is a general classification for the latter into two main types: inspection methods and empirical methods (Fernandez et al. 2011). With inspection methods, expert evaluators perform conformance reviews and problem prediction based on heuristics. Thus, inspection methods are limited in the kind of problems that can be found in a lab setting, the quality of the heuristics and the expertise of the evaluator. The limitations of inspection methods have led to the popularity of empirical methods, particularly user testing, which capture and analyze real usage data (Rubin & Chisnell 2008). Our approach is based on remote user testing and the usage data captured are UI events.

In 2000 Hilbert & Redmiles defined a comparison framework for characterizing automated usability evaluation methods (Hilbert & Redmiles 2000). According to this framework, our approach would mostly fit the categories: *Transformation*, since it involves selecting, abstracting, and recoding event streams, *Analysis*, specifically counts and summary statistics and sequences detection, and *Visualization* of abstracted events (usability smells). In general, we provide an *Integrated evaluation support*, i.e., an environment with integrated support for flexible composition of various transformation, analysis, and visualization capabilities. Sequences of actions are also analyzed in some usability smells detection, fitting the *Sequence detection* category, but it is not a central part of the approach. Another very comprehensive survey was made by Ivory & Hearst in 2001 (Ivory & Hearst 2001). This survey presents a taxonomy of usability evaluation methods according their automation level, and classifies a total of 132 evaluation methods in 4 different dimensions: Method Class, Method Type, Automation

Type and Effort Level. In this early work, the authors notice the many advantages of automating usability evaluations, but also highlight that a full replacement of manual methods is not possible. According to this taxonomy, our approach fits in the Method Class: *testing*, Method Type: *log file analysis*, Automation Type: *critique* and Effort Level: *minimal*.

2.3 Logs Analysis and Visualization Methods

The idea of gathering usability metrics from UI events has long been regarded as a valuable source of information, even beyond the web realm. Back in 2002, when the industry of web analytics was rising, researchers like Chi devised sophisticated data analysis techniques and visualization tools to help experts understand the large amount of data from user logs (Chi 2002). His visualization approach included site structure based on usage, traffic patterns, and path prediction compared to actual paths. In a similar way of current web analytic tools, Chi's visualization approach was able to point to navigation deficiencies of a website, though it required an expert web analyst to discover other concrete usability problems. Nowadays, web analytics services are provided by commercial tools like CrazyEgg[1] or Hotjar[2] that automatically gather interaction data from live websites, offering *heatmaps* and other usability-specific statistics. These services demand fewer resources, since end-users are the ones who unknowingly provide the required feedback. However, concrete, high-level problems are still hidden behind the statistics, and require a usability expert to uncover them and find their solutions.

Regarding the study of interaction logs, the work of Apaolaza et al. (Apaolaza et al. 2015) analyze low level logs captured "in the wild", i.e. from real users in real websites performing free tasks. They transform these logs into higher abstraction entities called *micro-behaviors*, and use them to demonstrate how users' behavior evolves over time. In a similar way, Breslav et al. (Breslav et al. 2014) propose to discover problematic *micro-interactions* by analyzing individual user sessions recorded in terms of interaction events. In our work, we also capture logs "in the wild", and take advantage from the abstraction of events in higher level entities, but we do not store or analyze full user sessions.

Other kinds of visualization methods for usability metrics has been studied in different approaches that perform remote usability tests. WebQuilt (Hong et al. 2001) is a proxy-based system for running task-based (controlled) usability tests with volunteer users. It captures visited pages and navigation paths, as well as time-based metrics, and shows the aggregated data from all test sessions in a zoomable visualization. Thus, while this tool provides helpful insights on user tests, it requires a usability expert to set up tasks, pinpoint usability problems and fix them. Atterer et al. proposed another approach to analyze logs during remote usability testing where usage data is collected on a proxy (Atterer et al. 2006). They capture interaction events transparently and in such detail that the output could be used for a number of different scenarios, from usability testing, to user profiling, and advanced visualization techniques for usage data. However, they don't expand on the processing required to use the output data in those scenarios.

[1] www.crazyegg.com/
[2] www.hotjar.com

In a similar way of that work, our tool captures interaction events transparently, though it gathers enough detail to help detect a usability smell, thus reducing processing time and complexity.

In a similar line of work, Okada and Fujioka proposed a method to detect usability problems from mouse traces (unnecessary and missed operations), by automatically comparing user logs with optimal logs of clicked points, gathered from tasks performed in an optimal way (Okada & Fujioka 2008). The authors report that the method can find 61% of the problems found manually in a much shorter time. The difference with our approach is that we do not require previous reference data. We also search for other events besides mouse traces.

2.4 Remote User Testing

Current tools in the area of remote user testing are WUP, WELFIT and CrowdStudy. Similarly to WebQuilt, WUP (Web Usability Probe) (Burzacca & Paternò 2013; Carta et al. 2011) uses a task-based approach for remote user testing, though in this case it logs events in the client. WUP uses sophisticated visualizations to show event log data: a timeline comparator allowing the interactive manipulation of timelines to find deviations with the optimal one and displaying color-coded events and associated data; a page-flow comparator or storyboard, and even screen dumps. In a more recent work, the authors have presented an improved, more interactive tool for timeline comparison (Paternò et al. 2016). The difference with our approach is that it works with a specific evaluation task list and the definition of an optimal path for each task. Moreover, while it provides automated techniques to discover deviations from the optimal path, it still requires a usability expert to discover concrete usability problems in the deviations, and their solutions. In the case of WELFIT (Santana & Baranauskas 2015), event logs are visualized through usage graphs. In a similar way of WUP's, it requires an expert to find deviations in walks present in the graph and thus discover usability problems in a potentially large graph.

Another work for organizing remote user tests and gathering results is CrowdStudy (Nebeling et al. 2013a). CrowdStudy presents an interesting approach to remote user testing since it incorporates crowdsourcing techniques such as Amazon's Mechanical Turk to recruit volunteers, and it also considers their context, like the type of device. They also present a set of metrics to gather the tests' results. Although being a very different approach from ours, since it's based on user testing with volunteers and guided tasks, the architecture of CrowdStudy is of interest to our proposal, since it also aims at gathering usability metrics from the largest possible amount of users.

2.5 Automated Log Analysis and Evaluation

We next report on three approaches for usability evaluation in uncontrolled environments that are closer to our proposal. Testing in uncontrolled environments requires different analysis techniques, since there are no optimal event streams to compare with, and the amount of data is potentially large and disorganized. Nevertheless, it has at least two benefits over controlled methods: first, they are less expensive as they do not require recruiting volunteers nor designing tasks, and second, they may find other problems that only appear in a real, uncontrolled environment, where users are not restricted to a limited set of tasks.

One of these approaches by Speicher et al. (Speicher et al. 2015) present a suite of tools for obtaining usability scores about Search Engine Results Pages (SERPs). Despite being focused only on SERPs, their approach also tracks UI events to measure usability and provides suggestions for improvements from a catalogue of best practices. Once a new version of a SERP is manually created, a comparative analysis of the old and new versions is performed through A/B testing.

The second work similar to ours is that of Harms et al. (Harms & Grabowski 2014), which also use the concept of usability smell to point to high-level problems in specific webpage elements. They use a slightly different definition of usability smell, by referring to them as *exceptional user behaviors that may indicate a usability issue*. There are two differences with this approach: first, the shape of the captured logs. They obtain long tasks from full user sessions and generate task trees to detect the smells, whereas we capture smaller grained events sequences (that do not compromise user privacy), since we aim to provide results in real time as soon as the smells appear. The second difference is derived from the first, and consists in the nature of the smells, which are more oriented to discovering inefficient tasks' flow. Besides their differences, both approaches described above could provide good complements for our ideas since the detection processes are quite different, and hence detect different kinds of problems.

The closest approach to ours in terms of functionality is, to our knowledge, W3Touch (Nebeling et al. 2013b). W3Touch is a toolkit devised for automatically assessing touch-based interaction problems on mobile devices and providing adaptations to solve them. It includes complex mechanisms to track touch events and touch-specific metrics, which by default are missed links and zoom levels. W3Touch provides default adaptations for some cases that consist in altering the size and spacing of the affected widgets, depending on the device, and allows triggering adaptation created by developers. The toolkit analyzes tracked data and identifies critical components as DOM elements that probably have touch related problems (missed links or zoom levels beyond some threshold values). This analysis may be compared to the processing that our usability smell finders perform when classifying DOM elements with potential usability problems. However, while W3Touch uses an ID-based classification of DOM elements to segment the page into critical components, USF uses a similarity algorithm to classify DOM elements in terms of semantic equivalence, and thus, in terms of having the same smell and the same solution. The main difference with our proposal is that W3Touch is essentially focused on responsiveness issues for touch-based devices, rather than usability in general. Also, W3Touch is able to detect interaction issues on single pages, while our approach is able to catch higher level usability smells that could entail navigation across pages, yielding to a broader range of detectable issues.

3 Usability Smells of User Interaction

This section provides the definition of usability smell that we use in this work, and describes our catalog of usability smells for user interaction. In the refactoring jargon, the term *bad smell* describes a potential problem with known consequences, and also known solutions by means of refactoring (Fowler 1999). In the case of usability, a bad smell is a hint of poor interface design that makes it difficult for the end-user to accomplish common tasks.

In our previous work we found that detecting usability smells could be very demanding as it requires experience and time. Therefore, it is a task that can be greatly improved with automation at different levels, which is why we strive a fully automated detection of usability smells. This would make the approach practical and useful for all developers, independent of their expertise in usability. By achieving automated detection of usability smells, our goal is to also point to a concrete usability refactoring to solve it. However, finding matches for smells and refactorings requires the abstraction level of usability smells to be much more concrete than the ones featured in our existing catalogs. For example, the smell *User Confusion* may have several unrelated causes like misleading links, lack of information or content redundancy, and thus, it is related to different refactorings as solutions, like *Improve the description of process links*, or *Clearly describe errors in executed activities* (Distante et al. 2014). Consequently, to build automated tools we had to define much finer-grained smells than in previous catalogs.

Moreover, since we advocate for agile development methods and incremental usability improvement, we aimed at incorporating constant user feedback by capturing real user events. Building upon previous literature on UI event log analysis, we defined a catalog of usability smells for user interaction by abstracting patterns of user events that were shown to create problematic interaction, and then linking those patterns to usability smells that can be (*i*) automatically discovered and (*ii*) solved by applying usability refactorings.

We next present a catalog of usability smells of user interaction. Each smell is described by name, description, and example. Afterwards, Table 1 connects each smell with its recommended solution in terms of a usability refactoring, and also the related smell from the previous catalog.

1) Undescriptive Element: this smell appears when many users try to get a tooltip from a webpage element. This may indicate that the element is not self-descriptive enough. According to the Design and GUI Guidelines for Windows 10 Developer[3]: "*A valuable tooltip will clarify an unclear action.*" An interesting feature of this smell is that it will appear no matter whether the tooltip is actually implemented or not, since the mere necessity of displaying can indicate a problem with the element's description.

For example, Figure 1 shows a screenshot of Google Drive where we found that inexperienced users hover over their names (in this case, "Alejandra") looking for a "log out" option. User names were removed from the heading of Google Drive in the latest version.

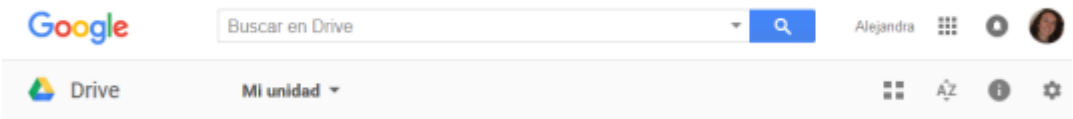


Fig. 1. Screenshot of the heading of Google Drive in May'2016

2) Misleading Link: this smell is similar to *Undescriptive Element* but specific to links. Besides tooltip attempts on links, this smell also captures sequences of events where many users navigate a link only to return shortly after, indicating that most probably, the linked contents are not correctly represented by the link name.

[3] https://msdn.microsoft.com/en-us/windows/uwp/controls-and-patterns/tooltips (last accessed Jun 8, 2016)

This problem with Misleading Links has been documented in the motivation for applying the web design pattern "Descriptive, Longer Link Names" (van Duyne et al. 2006), which points to the frustration of users that continually "pogo" back and forth when navigating links that don't interest them. A common example is a link poorly described by a stub text like "click here", which can lead to a content that the user did not expect.

3) No Processing Page: this smell is detected when users have to wait too long for a page to load without feedback. The waiting threshold intends to capture the moment when users lose interest. This has been studied in the literature, and although the thresholds vary from one study to the next, they typically range between 8 and 15 seconds (Nah 2004). During our trial runs we have achieved best results with a 10 second threshold. This smell is inspired by the motivation for the web design pattern "Processing Page"[4].

4) Free Input for Limited Values: this smell is triggered when a standard text input is used for entering data from a limited set of values, like cities or countries. Users are forced to type the full text when in fact the options are restricted. In the case when there are few options, a select box is more suitable than a text input, and when the number of options drops below 5 or 7, a radio button set is more appropriate. This smell may also appear when the options are not restricted but there is actually a narrow set of very popular choices. In that case, at least an auto completion feature should help users type less text, feeling more reassured about the entered option.

5) Unformatted Input: this smell indicates that a standard text input is being used to enter data that has a determined format, like dates, phone numbers or zip codes. In these cases, the form could help the user format the data. While entering phone numbers or dates, for example, automatic formatting can be very useful, especially when the site has an international audience. In our experiments we found that using calendar widgets instead of unformatted text inputs for dates may improve satisfaction in use in online stores (Grigera et al. 2016). This smell is detected by matching the entered data with regular expressions (one for each format).

6) Short Input: this smell is detected when an input field is not wide enough to match the size of the most usually entered texts. Inadequate text input width is a confusing issue, as pointed out by authors like Wroblewski (Wroblewski 2008). In fact, text inputs that are much wider than actually required can also confuse users (we plan to capture these cases too).

7) Unnecessary Bulk Action: it is usual for web applications that display lists of items (like products or messages) to offer bulk action capabilities. Users typically perform these actions by first selecting a group of items using checkboxes, then selecting the action, e.g. "Delete", or "Move to…" and finally applying the action (although this last step might be unnecessary, depending on the way of choosing the action). While this interaction works well when applied to a group of items, it takes unnecessarily long to apply to a single one. When the most frequent case is the latter, and users do not have (or do not use) an alternative, faster, way of applying

[4] http://welie.com (last accessed Jun 8, 2016)

these actions, the smell Unnecessary Bulk Action is detected. This smell emerged from our own previous work on accessibility refactoring, and appeared as "Difficult interaction to apply operations on elements" (Garrido, Rossi, et al. 2013). For unsighted users, this smell is quite serious, as it requires going through the page elements many times with the screen reader, first to read the list elements, then to select one or more, and finally to select the operation to apply on the element, which usually appears at the top of the list.

8) Overlooked Content: some content is usually scrolled-through very quickly by users, so it is probable that they don't read/see it. Also, they usually stop at the same area (*y-position*), so it's likely that the reached content could be easier to find if it were closer to the top. While nowadays users are allegedly more willing to scroll, they still need a reason to do so, since it still involves an extra interaction step. Apart from signaling a potential usability issue, this smell provides valuable input when the overlooked content is considered of importance by the web application's owner. Some web analytic tools provide scroll maps which show how far down the page users scroll, but do not show at which speed.

9) Distant Content: this usability smell detects repeating navigation paths where users only stay for a brief time at all intermediate nodes. The rationale behind the smell is that users are seeking for the contents in the final node, but this requires a long path to reach. A usual way of studying navigation usability is often by comparing optimal paths with real ones followed by users, either in a real context or in experiment settings (Atterer et al. 2006). The Distant Content smell, however, indicates that the most followed path from node A to node B might be too long.

10) No Client Validation: reporting errors on web forms in a clear and timely way is a widely known guideline for web forms usability (Seckler et al. 2014). This smell detects forms that have a high rejection rate, where validation happens on the server side. This scenario is detected by keeping track of a new request, and the presence of the same form after the response.

11) Late Validation: this smell is essentially the same as the previous one, *No Client Validation*, with the difference that it is specific for the scenario where validation occurs after the "Submit" button is clicked, but submission does not happen. This usually means there is client-sided validation, but could be improved by inline validation. This kind of validation has been reported to improve several metrics like success rates and completion times in some cases [5] (Wroblewski 2008).

12) Abandoned Form: this smell is detected when the drop-off rate of a form exceeds a given threshold, thus alerting the evaluator of a form that is probably too complex and it discourages users to fill it. During our trial runs, we got better results with this threshold set to 40%, although this is particularly difficult to standardize since web forms have many different purposes; a login form is expected to have much higher success rate than a contact form or a registration.

[5] http://alistapart.com/article/inline-validation-in-web-forms (accessed Jun 14, 2016)

13) Scarce Search Results: search forms are a fundamental part of the web, as pointed out by Nielsen at the top of his "top 10 mistakes in web design"[6]. This smell is detected when a form does not bring results most of the times. It also saves the most popular unsuccessful queries to provide the web owner with a ranking.

14) Useless Search Results: as a variant of the previously listed *Scarce Search Results*, this smell detects what happens after a search form *does* bring results. If users rarely click on one of such results, then it might be that such results, even if they appear, are not what users were searching for. Sometimes, however, the results page can be informative enough that a further click is not necessary, and this smell could be ignored.

15) Wrong Default Value: filling up a form can be tedious, so having as few questions as possible increases the likelihood of users completing it. Having reasonable default values ("smart defaults" (Wroblewski 2008)) for the questions decreases completion time even more. This smells detects whenever the most popular choice for a radio button set or a select box doesn't match the default value. This smell is linked to the smell *Unnecessary activities in the main process* from our previous catalog.

16) Unresponsive Element: this smell is detected when an element is usually clicked by users, but does not trigger any actions. This happens when such elements give a hint because of their appearance. Typical elements where we have found this smell include products list photos, website headings, checkbox/radio button labels, underlined/highlighted texts.

As we previously mentioned, most usability smells presented in this section are specializations of more generic ones presented our previous catalog (Distante et al. 2014). In Table 1 we include a list of all refactorings and, when applicable, also link them with the previously catalogued smells.

---

[6] https://www.nngroup.com/articles/top-10-mistakes-web-design (accessed Jun 14, 2016)

Table 1. List of usability smells and associated refactorings

Smell Id	Usability Smell	Abstract Usability Smell(s)	Refactorings
1	Undescriptive Element	User Confusion	Rename Element / Change Widget
2	Misleading Link	User Confusion	Rename Anchor
3	No Processing Page	Premature Abandonment	Add Processing Page
4	Free Input For Limited Values	Risk of Error / Activity too Long / Frequent Empty Results	Add Autocomplete / Change Widget
5	Unformatted Input	-	Change Widget
6	Short Input	User Confusion	Resize Element
7	Unnecessary Bulk Action	Activity Too Long	Distribute Menu
8	Overlooked Content	-	Split Page / Remove Redundant Content
9	Distant Content	User Distractions	Add Link
10	No Client Validation	Subsequent Failed Validations	Anticipate Validation
11	Late Validation	Subsequent Failed Validations	Anticipate Validation
12	Abandoned Form	Premature abandonment / Activity too Long	Split Activity / Postpone activity
13	Scarce Search Results	Frequent empty results.	Add Autocomplete
14	Useless Search Results	-	Add Autocomplete
15	Wrong Default Value	Unnecessary activities in the main process	Set Default Value
16	Unresponsive Element	Difficult Access to Information / Absence of Meaningful Navigation Links	Turn Attribute Into Link / Change Widget

The definitions of refactorings on the right column of Table 1 appear in previous works (Garrido et al. 2011; Garrido, Firmenich, et al. 2013; Distante et al. 2014; Harold 2008). Some refactorings, however, were catalogued in more general forms than the ones in Table 1: *Rename Anchor* is a concrete form of *Improve the description of process links* and *Resize Element* is a specialization of *Change Widget*. Moreover, note that these refactorings were catalogued as solutions to more abstract usability smells. For example, the general smell *User Confusion* may be solved by any of the refactorings *Rename Element*, *Rename Anchor*, *Add Processing Page*, *Split Page* or *Remove Redundant Content*. Thus, this work contributes to a more precise matching between usability smells and associated refactoring.

It's important to remember that usability smells are hints or indicators of problems, and in some cases they could not point to an actual usability issue, depending on the context. For instance, a *Wrong Default Value* could suggest that the most popular choice should be set as default, when in fact the designer requires a different one (e.g. when setting on purpose the option to receive a newsletter).

4 The Process

Our automated strategy to usability smell identification is based on a process consisting of three steps: **Events Logging**, **Usability Smells Detection,** and **Reporting**. In addition, the architecture for this process is divided into a client-side component that performs Events Logging, and a server-side component that is responsible of the remaining two steps.

In the **Events Logging** step, the client-side component mines fine-grained events to filter and aggregate the relevant ones into mid-level events called *usability events*. In the **Usability Smells Detection** step, the server-side component classifies and analyzes usability events using specialized algorithms to discover usability smells. Finally, the server-side component is also responsible for the **Reporting** step, when usability smells are instantly visualized along with the suggested refactorings that may solve them, and with enough detail for developers and stakeholders to understand the problems and take action to fix them.

Figure 2 shows an overview of the 3 steps of the process and the following subsections describe each step in detail.

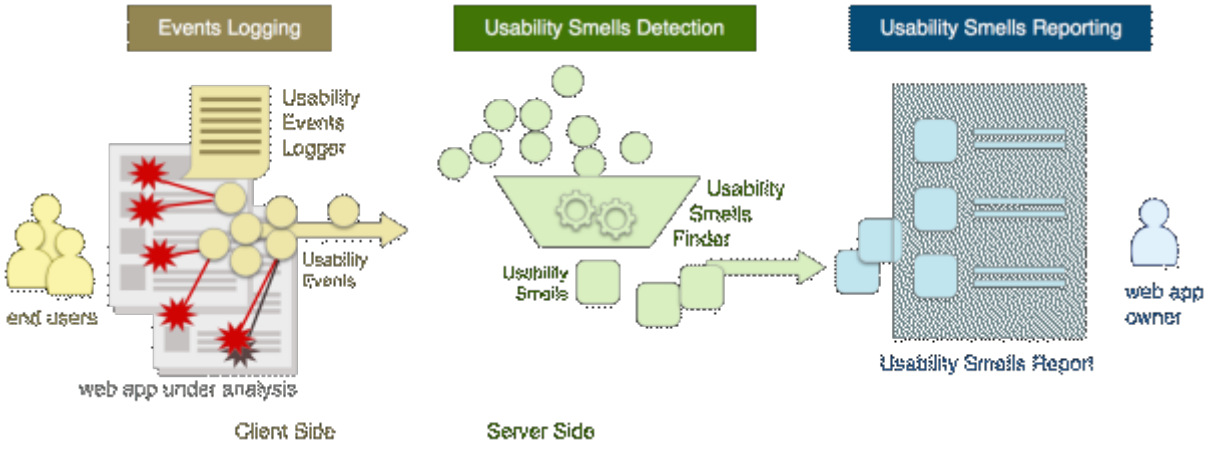


Fig. 2. The three steps of the process

4.1 Events Logging

Events logging is a strategy that many researchers have long studied to understand user behavior and problems in user interaction (Ivory & Hearst 2001). Most approaches simply log all UI events at the client, and later perform sophisticated log analysis at the server, to provide visualizations of different metrics or compute deviations of user events from an optimal way of performing a task.

During our own research in web usability refactoring, we realized that manually detecting usability smells on web applications was a costly task that could be automated by using known analysis techniques over UI events. It is important to note, however, that we aim at providing developers with real-time reports as soon as the smells are discovered, and processing potentially huge amounts of low-level logs on the server would take too long. For this purpose, we perform a preliminary analysis at client-side, filtering and aggregating low-level UI events and sending to the server only the ones that could lead to detect a usability smell. Besides, by doing client-side processing we are able to gather information of the user context at the very

moment the event is taking place. Thus, the aggregation of low-level UI events, together with context information about the event, allow us to generate higher-level events that we call **usability events**.

Usability events were mostly designed in a top-down fashion from usability smells, i.e., we first observed the behavior that indicated the presence of a usability smell, and then decomposed this behavior in smaller-grained actions that can be automatically captured from single users in single sessions of work. An example of a usability event is *Flash Scroll*. When a user scrolls faster than a given speed, a *Flash Scroll* event is logged that aggregates all relevant data like starting/ending vertical position, speed and timestamp. All usability events generated by different end-users are collected and analyzed at the server, in the second step of the process. In this example, if enough users trigger the Flash Scroll event on the same DOM elements, this will indicate the presence of the usability smell *Overlooked Contents*, that is, the often scrolled-through content may not be of interest to the visitors.

There is a many-to-many relationship between usability events and usability smells, i.e. a single usability event can help to detect different kinds of usability smells, and detecting certain usability smells requires analyzing more than one kind of usability event. Below we describe all usability events and afterwards Table 2 lists the correspondence between usability events and usability smells. For each event we also define its threshold values.

For all events that apply to a DOM element, such element is stored including its xPath, dimensions and position, which are later used in the reporting stage to display it. The HTML code is also logged, since it is fundamental to cluster events by similarity as they arrive at the server. All the events are stored with a timestamp.

A) Tooltip Attempt: this event is generated when a user poses the mouse cursor on an anchor or image for a certain amount of time. The event includes the affected DOM element and the waiting time. There is a minimum and maximum waiting time threshold, set by default to 1.5 and 5 respectively, which were the values that better captured this behavior during our preliminary experiments.

B) Click Attempt: this event is generated when a user clicks on an element that doesn't react. Mouse clicks on most DOM elements (excepting e.g. anchors or form controls) are tracked, and if they are not followed by a navigation or an action, the event is triggered. A Click Attempt stores the timestamp and information on the clicked DOM Element. The algorithm that detects this event rules out clicks made on large elements, since they usually represent accidental clicks that, for instance, users make to ensure focus on the browser's window. It also discards clicks made for selecting text, or dragging elements.

C) Flash Scroll: this event is generated with a quick scroll action, either up or downwards, over a long area. The data that is saved with the event is the URL, initial/final top, and duration of the action. Based on our experiments, the default scroll speed threshold is set to the page height divided by 2 seconds.

D) Flash Navigation: this event consists in a rapid navigation sequence, where a user navigates to a given page, browses it for a very brief time, and returns to previous page shortly after. The

event stores the DOM information of the clicked anchor, both source and target URLs, along with the dwelling time at the target. The threshold for maximum dwelling time is configured to 3 seconds.

E) Navigation Path: this event is similar to *Flash Navigation* in that it reveals rapid navigation sequences, but in this case the event is created when 2 or more quick navigations happen in a row, over a path of pages. The path is recorded, where each node contains the clicked anchor and the target URL. The dwelling time at each node is set to 3 seconds.

F) Search: The *Search* event is used to count the number of successful/unsuccessful searches that users perform, and whether successful searches were actually useful. The algorithm that detects this event follows 3 steps:

1. When the user submits a form, the first step uses a simple heuristic to determine whether it is a search form (by analyzing properties like the submit button's caption, the number of text inputs, the word 'search' in different languages). In the case of a search form, the search query is stored.
2. After the submission, the second step scans the resulting page for a list of results that contains the search query. This step attempts to establish whether the search was successful or not.
3. Once the user leaves the results page, the third step logs how that happened, i.e., whether the user clicked one of the results (meaning the search was useful), or navigated away.

Only after the third step is the event sent to the server, with all the associated information: search query, search form contents, presence of search results, and whether the user clicked on any of them.

Since steps 1 and 2 are based on heuristics, we adjusted them based on several tests on popular sites to ensure they cover the most possible cases.

G) Bulk Action: this event is logged when a set of elements in a list is checked, and then a submission occurs. The algorithm covers different scenarios using the following heuristics:

- to detect a list of *checkable* elements, it searches for a list of aligned checkboxes - or elements shaped as checkboxes with attributes "class" or "role" equal to "checkbox". The minimum number of elements to be considered as a list is configured by default to 4.

- to detect a bulk action, the algorithm verifies if no checkboxes are checked after any click or navigation.

This event logs the form's DOM information, along with the count of checked items.

H) Option Selection: this event logs a selection made on a select box or a radio button set. It particularly detects whether the default option has been changed. This information is logged along with the selection index (and also text, in select boxes) and the form's DOM element data.

I) Form Submission: the main aspect captured by this event is the validation strategy in a given form, i.e., at which point after a failed form submission errors are displayed to the user. The algorithm for this event evaluates what happens after the submit button is clicked. If a new navigation follows, then it searches for the same form on the destination page, in order to

determine if the submission was rejected (excluding search forms, since they usually appear back again in the results' page). If no navigation happens, then the form is assumed to have client-side validation. The validation strategy is logged along with the form's DOM element data.

J) Unfilled Form: this event is generated when a form gets partially filled and then abandoned.

K) Text Input: this event is logged any time a user fills a text input, along with several additional details:

- input time: the amount of time the user spent writing, i.e., from *focus* to *blur* events.
- keystrokes count: amount of keys pressed, which could be different from the resulting text size. This can be used for deriving interesting insights, e.g. when it is usual to have few keystrokes and much larger texts, it may indicate the presence of auto-completion.
- resulting text (except when the input type is "password")
- correction: indicates whether the input was previously populated
- DOM element information.

This event in particular provides useful information to the process and helps to detect several usability smells.

L) Long Request: this event logs whenever a request takes longer than a given threshold. The event logs the source and target URLs, along with the waiting time.

Table 2. Usability Events and their related Usability Smells.

Event Id	Event	Usability Smell
A	Tooltip Attempt	Undescriptive Element
B	Click Attempt	Unresponsive Element
C	Flash Scrolling	Overlooked Contents
D	Flash Navigation	Misleading Link
E	Navigation Path	Distant Contents
F	Search	Scarce Search Results, Useless Search Results
G	Bulk Action	Unnecessary Bulk Action
H	Option Selection	Wrong Default Value
I	Form Submission	Late Validation, Abandoned Form, Scarce Search Results, Useless Search Results
J	Unfilled Form	Abandoned Form
K	Text Input	Free Input For Limited Values, Unformatted Input, Short Input
L	Long Request	No Processing Page

4.2 Usability Smell Detection

This step consists in classifying, aggregating and analyzing the captured usability events to discover usability smells. Since the analysis for detecting different smells varies from one to the other, there is a different algorithm for each one. These algorithms are implemented in entities called **usability smells finders**, and each one of these finders is able to detect a single kind of **usability smell** by consuming and analyzing one or more kinds of **usability events**. Each application under analysis has its own set of finders, and each finder is configured with certain parameters that determine the number, proportion, or combination of usability events that trigger a specific smell. Even if we provide default optimal values obtained through experimentation with the tool, they can be adjusted for specific purposes.

In order to produce results in real time, usability events are processed at the moment they arrive at the server, and all the individual usability smells finders re-evaluate the events to detect potential new usability smells. To speed up the detection process, we adjusted it so the execution time does not depend on the amount of logged events. This happens in 3 steps:

1. **Events Classification:** at the time of arrival, the new usability event is sent to the associated finder(s). For example, if a new Click Attempt arrives, it is sent to the finder that detects **Unresponsive Element** smells. Once the event has reached a finder, it is classified once again, this time depending on the element it affects (e.g. a DOM element). This allows us to re-evaluate only the events affecting this particular element.
2. **Data Synthesis:** as soon as the event is classified by its affected element, the finder extracts some key information from it. For example, it might increment a count or update a buffered average. This synthesis step is key for the process' performance, since it avoids going through all the events each time a new one arrives.
3. **Usability Smells Evaluation:** with the updated info brought by the new event, the single finder reevaluates the presence of a smell on the affected element. New smells will be added to the report, and old smells which are no longer detected (because updated values do not meet the thresholds anymore) are kept for completeness of the report, but this is indicated to the evaluator.

This way of processing data, especially the Data Synthesis step, allows us to obtain instant results without depending on the amount of usability events. The three steps for usability smells detection are depicted in Figure 3. In the figure, circles represent usability events that travel from the client to the server component, where they are clustered to help detect usability smells.

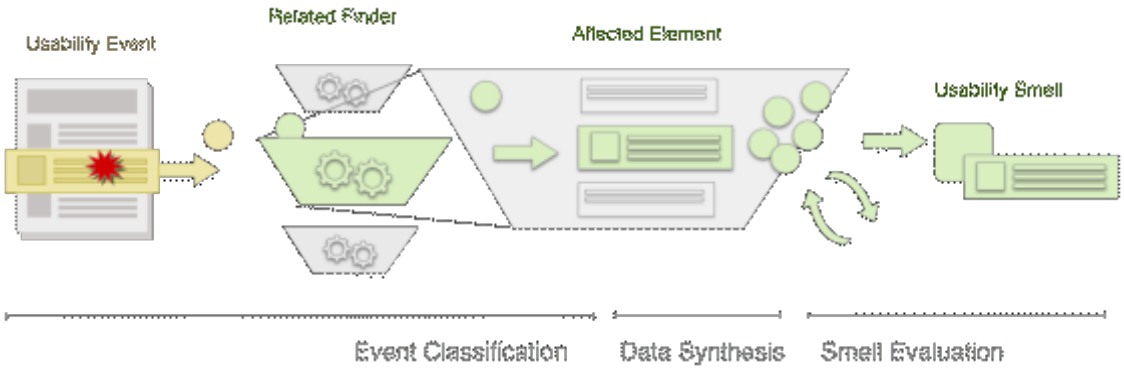


Fig. 3. Usability Smells Detection

The set of finders can be easily extended. The architecture is prepared in a way that adding a finder for a new kind of usability smell only implies selecting the kind(s) of usability events to be consumed, the criterion to group them together (by default is set to same/similar DOM element), and the logics for detection, that typically involves counting proportions on the grouped events' properties.

We will illustrate the detection process with two usability smells, *Scarce Search Results* and *Free Input for Limited Values*, showing a brief description for each one, the usability events that trigger them, and how these events are processed. Additionally, we enumerate the parameters that can be adjusted for the detection process and the refactorings suggested to solve the usability smell.

Scarce Search Results

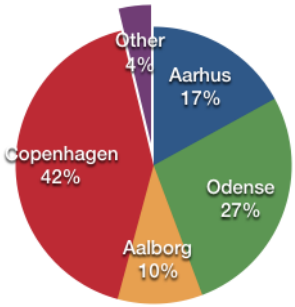
- Description: a search form usually fails to bring results.
- Associated Usability Events: Search.
- Detection Process:
  1. For each form affected with search events, split the events in two groups: those that indicate results, and those that indicate absence of results.
  2. If the group of events without results is proportionally larger than the other set by a given threshold, then trigger the bad smell on the form.
- Adjustable Parameters:
  - Minimum "no results" proportion: *what is the minimum proportion of searches with no results to consider the presence of the bad smell?*
- Recommended Refactorings: *Add Autocomplete* to the search form, decreasing the users' uncertainty before hitting the *search* button.

In this case, the logics are simple, but the client-side script plays a major role at collecting usability events, since it logs whether the search action was successful or not.

Free Input for Limited Values

- Description: a free text input is presented to the user, but the accepted values belong to a limited set.
- Associated Usability Event: Text Input.
- Detection Process:
  1. For each input field, collect all Text Input events.
  2. For all Text Input events on a given input field, group together the events with a same or similar value. By default, the finder uses Levenshtein's algorithm to compute string similarity (Levenshtein 1966). Collect the remaining unique values in a group called ***other***.
  3. Count the events for each group.
  4. If *(a)* the size of each group except ***other*** is larger than a minimum proportion threshold, and *(b)* the proportion of the group ***other*** is smaller than a tolerance threshold, then trigger the bad smell on the current input field.
- Adjustable Parameters:
  - Minimum events count: *how many Text Input events an element must have before considering the bad smell detection?*
  - Minimum group proportion: *how large must a group of similar values be, in terms of proportion, to consider it a frequently typed value?*
  - Maximum proportion of unique values: *how many infrequent values should be tolerated?*
  - Similarity algorithm: *how do we determine that two different values must be considered to be the same?*
- Recommended Refactorings: There are two possible solutions for this bad smell. One is *Add Autocomplete* for the input field. The other is *Change Widget*, i.e. replace the input field by a fixed selection widget such as a select box or a radio buttons' group, depending on the number of options. Note that with the second solution, it is still important to offer a generic option "other", even if there weren't infrequent values observed, so to preserve the original functionality that allowed any value to be entered.

For example, consider a checkout form where users indicate a shipping address, and the "City" field is a text input. The website only operates in Denmark, so there is a limited set of possible cities. Analyzing the events, the system finds large groups of repeated values like "Aarhus", "Copenhagen" and "Aalborg", even with typos or different casing/spelling (e.g. "ÅRHUS"). Figure 4 shows this distribution of values.



Values entered at //@city input

VALUE	TIMES ENTERED
Aarhus	400
Odense	640
Aalborg	234
Copenhagen	987
Other	89

Fig. 4. Sample data for detection of Free Input for Limited Values

While there are also a 4% of values with random text or unknown cities, the "City" field is affected with the bad smell *Free Input for Limited Values*.

4.3 Reporting and Refactorings Suggestion

The final step of our process involves reporting the detected usability smells, along with the refactorings that can be applied to solve them. At this stage, each smell is presented with information intended to help the developer make a decision. For example, the *Scarce Search Results* smell informs a ranking of the top queries that brought no results to the end users, and the *Late Validation* shows the bounce ratio of the form. As we previously explained, in the particular case that an existing smell is no longer detected, the smell is kept in the report with an indication for the evaluator, for completeness purposes.

The report also recommends the refactorings that may solve each detected smell. When the developer applies a refactoring, they should reset the statistics for that particular smell, so the process of smell detection starts again. This iteration causes a continuous assessment of the website, to ensure that the solutions applied actually solved to reported problems.

5 The Tool

We implemented our approach in a tool called **Usability Smells Finder** (USF), which works as Software-as-a-Service. Usability evaluators can sign up, and the tool provides a code snippet that they must have included in their web application. This snippet enables the tool to log interaction data for finding usability smells, which are then graphically reported in the website owner's account. An early version of USF tool was introduced in a previous publication (Grigera et al. 2014).

5.1 Event Logging Mechanics

The Events Logging step is implemented by a client-side script that intercepts selected low-level UI events. Whenever an end user loads a page, the script starts analyzing low-level events like single keystrokes or mouse moves. It then processes these events according to different criteria and generates higher-level usability events as described in Section 3, which are sent to the server

for further analysis. This client-side processing also discards events that will not help to detect bad smells, reducing the communication with the server and preventing unnecessary heavy-weighted computation. The script is able to capture usability events even when they require information across different pages (or requests), such as the Navigation Path event, which keeps track of quick navigations sequences. Still, it does not interfere with the users' interaction, making it suitable for production stage. Threshold values for usability events generation may be adjusted in the snippet code. This way expert evaluators may tweak the tool to their needs.

When developing a tool that logs events from users, their privacy becomes an important concern. We implemented some measures to mitigate the risk of user identification from the logged information. First, the UI events our tool logs contain no information about the users or their devices (such as IP addresses). While the script is able to keep track of events even across different pages, these events are only focused on simple actions instead of complete tasks, and there is no reliable way of reconstructing a particular user's session by grouping events. In addition, the client-side script never records passwords or credit cards, only a stub text for the sake of indicating that something was typed. In the case of passwords, the tool verifies the input's type; in the case of credit cards, after a first detection of input name and label, the script verifies the entered number using regular expressions. Notice that only the format is checked, and other validations like the Luhn algorithm are omitted, since storing an *almost correct* credit card number (or many similar ones) can be nearly as dangerous as storing the correct one. No matter these basic precautions, there is still a risk of user identification, and further anonymization techniques should be used to intensify privacy. For example, emails and phone numbers could be detected and substituted with data masking techniques to preserve a real looking data. Moreover, in the cases where storing a realistic data format is not useful, e.g. people's names, this data could be directly obfuscated as done with passwords. However, in this case it is important to correctly detect the data nature, so valuable, not private information is not kept from the tool.

5.2 Smell Detection Mechanics

The Usability Smell Detection step takes place at the server, using the different kinds of usability smell finders. Each kind of finder detects a specific type of usability smell, and each application under analysis has its own set of finders with potentially different configurations. As previously described, we provide a default configuration to parameterize finders, but evaluators may adjust the configuration to their needs.

A finder classifies the detected usability smells by a common criterion, generally by the affected DOM element, but also by URL (smells *No Processing Page* and *Overlooked Contents*) or URL sequence (smell *Distant Content)*. When smells are classified by DOM element, it's important to note that a single HTML template may be used to generate different but equivalent DOM elements, i.e., different instances of the same widget (for instance, different products in a list). Detecting these cases is key to our approach, to prevent populating the usability smells' report with different smells that are actually a single one on equivalent elements. The tool is able to establish the level of similitude between two DOM elements, to determine whether they share the same template. This is achieved by using a tree similarity algorithm, which determines the edit distance between two DOM elements' trees, in a similar way than the Levenshtein

distance does for comparing strings (Levenshtein 1966). This is used along with a simple clustering strategy to classify the DOM elements. We have in fact implemented different comparison strategies in the USF tool, by modifying known Tree Edit Distance algorithms like RTDM (Reis et al. 2004), usually conceived for analyzing full web pages, to compare smaller DOM elements, as required by our approach. We use a strict matching to avoid grouping DOM elements that are semantically different. Details of such implementations are out of the scope of this paper, but during early tests with a group of 350 DOM elements captured from known websites we have reached a precision of 0.963, and 0.992 recall in the DOM elements similitude comparison. At the time of writing this paper, we are still conducting experiments to find the best DOM matching algorithm in terms of precision and performance.

5.3 Reporting Mechanics

Our tool reports bad smells right as they appear. Despite that it requires a minimum amount of events before showing interesting results, evaluators can check on the application's bad smells status at any time. For enabling on-the-fly reports, we developed a web frontend that shows detailed information on every bad smell, including when possible a live view of the affected widget (or a representative instance). Figure 5.a shows a screenshot of a real application used for the *Smells Experiment*. Two problems of the web page are highlighted in the figure: the one labeled as "A" is a widget that displays the steps composing the current process and the current step being executed, but it does not allow to navigate back to a previous step; the one labeled as "B" is an error message that appears after hitting submit and the validation fails. Figure 5.b shows two screenshots of the USF tool reporting two usability smells: *Unresponsive Element* and *Late Validation*. In this case, the *Unresponsive Element* smell was detected after many users attempted to go back in the checkout process, by clicking on the previous steps buttons without realizing they were actually grayed out. The *Late Validation* smell was detected after many users tried to submit an incomplete form that didn't indicate the required missing fields until they hit "Submit".

Each usability smell shows specific data for a better understanding of the problem. In the case of *Late Validation*, the reporter shows the percentage of unsuccessful form submissions, and in *Unresponsive Element*, the number of times the element was clicked. The tool also suggests refactorings for the detected smells, together with a brief description and a link to "Learn more" about the refactoring mechanics. Evaluators can also ignore usability smells presented in the report (as shown in Figure 5.c), either because they do not consider the smell to be a problem or because of a false positive. Moreover, evaluators may reset the stats for a given smell, e.g., after applying a refactoring.

In Figure 5.b, the reporter suggests the refactoring *Turn attribute into link* (Garrido et al. 2011) to solve the smell *Unresponsive Element*, in this case by allowing users to go back to a previous step on the process. In the case of the *Late Validation* smell, it suggests the refactoring *Anticipate Validation*, which embodies validating the input fields earlier, to prevent the users from trying to submit before the right data is entered.

Fig. 5.a. Screenshot of a web application under analysis

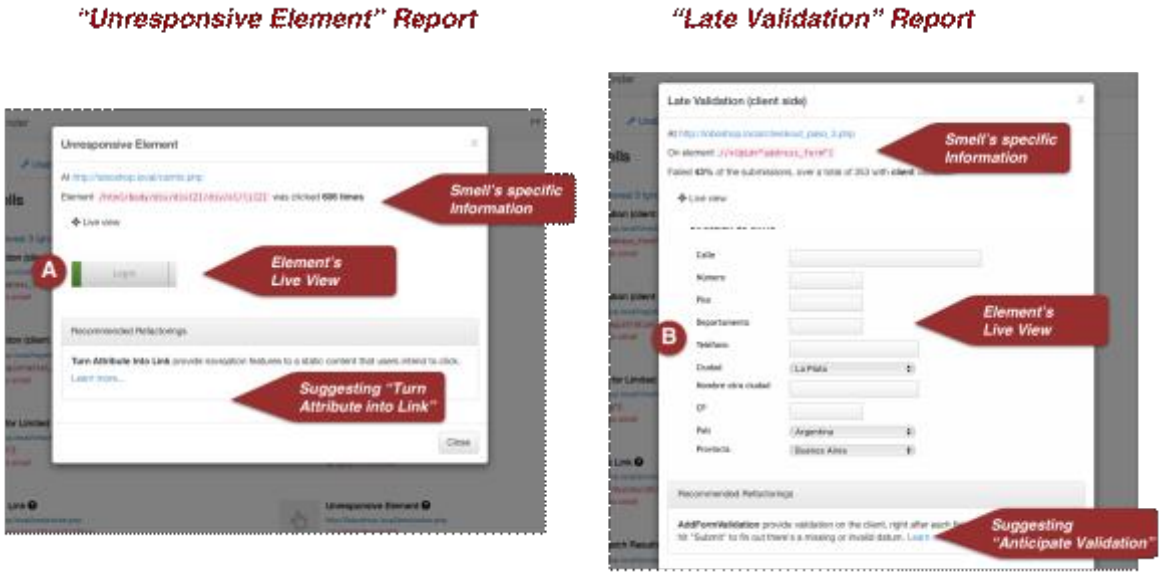


Fig. 5.b. Screenshots of the tool displaying captured usability smells

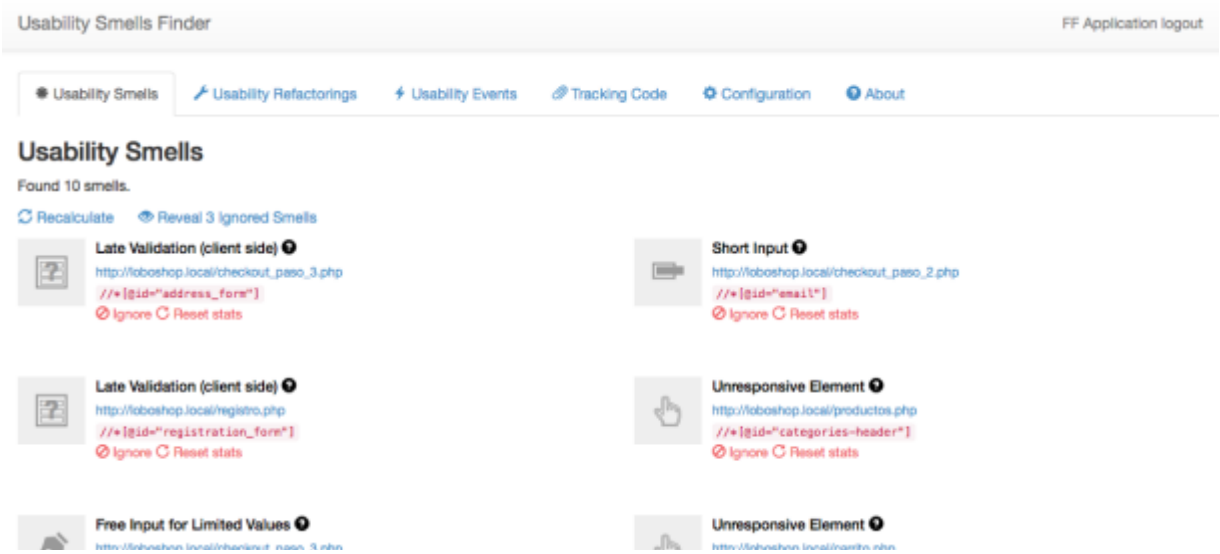


Fig. 5.c. Screenshot of bad smell report

6 Assessing the Approach

To assess the validity of our process and its implementation with the USF tool, we ran two experiments. In Section 6.1 we describe an experiment that validates the usability events detection process performed by the client-side component. In the remainder of the paper we will refer to this experiment as *Events Experiment*. In Section 6.2 we show a second experiment that validates the server-side component with respect to the success and reliability of the usability smells detection process. This experiment will be referred to as *Smells Experiment*.

6.1 Usability Events Validation

Our approach depends heavily on the correct detection of usability events. Some of these events involve heuristics that may be error-prone, like interpreting user's intention (*did the user actually mean to click on this widget?*) or inferring the nature of a GUI widget (*is this DOM Element a search form?*). We first performed a preliminary evaluation with several trial runs on different websites, and we were able to adjust the events detection heuristics and parameters. Following the preliminary evaluation, we performed an experiment to validate the accuracy, precision and recall of the USF tool when detecting usability events using these heuristics and parameters. Such experiment is described in this section.

To do this validation we actually had to consider two very different scenarios: on the one hand, we needed to test heuristics involving user intention interpretation, and on the other, those involving detection of DOM Elements. Other heuristics were very simple or involved no interpretation whatsoever, so we did not consider them for the validation. We detail all these cases in the following subsection.

6.1.1 Experiment Definition and Planning

Before actually planning the experiment, we determined that we had 3 different groups of heuristics within the 12 usability events the USF tool is currently able to detect:

1. **Heuristics involving user intention:** the usability events that depend on these heuristics are *Tooltip Attempt, Click Attempt, Flash Navigation, Navigation Path,* and *Flash Scrolling.*
2. **Heuristics involving detection of DOM element semantics:** the usability events that depend on these heuristics are *Bulk Action, Search,* and *Form Submission,*
3. **Low level heuristics:** these heuristics depend almost solely on low level events, so they are very unlikely to fail, or the few scenarios where they could, are very hard to reproduce in an experiment with a restricted set of web applications. Hence, we left them out of the validation. They are the following:
    a. **Text Input**: filling up a text field is a very low level event and we consider it very difficult to either miss it (false negative) or misinterpret it as an accidental action (false positive).
    b. **Option Selection**: this event detects when a user has selected an option from a select box or a radio button set. We didn't validate this heuristic since cases where non-standard widgets are used for these kinds of selection, while they do exist, are very unusual.
    c. **Long Request:** this particular event detects a long time between requests. After 3 seconds between page loads, the event is triggered. Although we do not rule out the chance of misdetection, during our tests we found no cases where this failed. It should be noticed that the 3-second threshold was set for the sake of not missing events, and the associated smell has a 10-second threshold, as explained in Section 2.
    d. **Unfilled Form:** this event combines the detection logics of Text Input (to detect the moment a user starts filling a form) with the low level *page unload* event, which can be securely captured in the vast majority of web applications. Hence, it has a marginal chance of misdetection.

To test the heuristics in group 1, we asked a group of users to perform specific interactions, and then compared the usability events we observed "manually" with the usability events automatically captured by our tool during that period. The experiment preparation then consisted in creating scenarios where these usability events should be detected, and a second set of scenarios where the events should not be detected but the tool could be "tricked" into doing so. We created different groups of such scenarios to generate more comprehensive tasks in order to recreate realistic situations for the subjects. We used 3 websites of different domains, and recruited 15 volunteers to run tasks on 2 websites each. Volunteers were 8 males and 7 females with ages ranging from 26 to 64 ($\bar{x}$ 35.66, s^2 75.52), and had diverse backgrounds and knowledge on the selected websites, from regular users to newcomers. Further details about users and tasks can be found in Appendix A.

During the test, we ran the tool to capture usability events, while we closely observed the interactions to manually detect events that the tool might have missed, or incorrectly logged.

Two evaluators were involved in the experiment: one was in charge of guiding the volunteer through the tasks while the other monitored the events as they happened using a modified version of the USF tool that provides instant visualization of usability events. While the first was concerned with the undetected events (false negatives), the second was in charge of detecting the ones incorrectly logged (false positives). In the cases where we used popular websites on which we naturally cannot install the client-side script, we incorporated it using browser extensions.

The heuristics in group 2 (for events *Form Submission*, *Search* and *Bulk Action*) are mostly based on DOM element analysis and need not be tested by different subjects, since their interpretation do not affect the results. Instead, we tested them across different websites to assess the tool's ability to detect the same situation across different pages. The evaluated websites were obtained from the Alexa's top sites ranking[7] (local and global), plus those from the experiments with volunteers. The criterion for selecting the sites was twofold: first, we had to make sure they involved some interaction, as opposed to text contents only. The other criterion was merely technical: given the technique we used to simulate the tool's installation by using browser extensions, part of the tool's functionality was in some sites blocked for security reasons, so we had to discard such sites.

6.1.2 Results

After the experiment for both groups of heuristics, we cross examined the observed results and the logged usability events to obtain different indicators. We used the F2 score to obtain a measure on the precision and recall of the tool's performance. We selected this measurement over the most widely known F1 score because it gives recall a higher weight, and at this stage of the process we prioritize recall over precision - the second stage of the process may dismiss false positive events when finding smells, but missed ones can't be recovered. Table 3 shows all measures for each usability event.

Table 3. Results of the experiment on usability events capture

	Accuracy	Recall	Precision	F1 score	F2 score
Click Attempt	88.00%	97.37%	88.10%	0.94	0.97
Tooltip Attempt	57.89%	82.22%	60.66%	0.70	0.77
Search	68.57%	71.88%	92.00%	0.81	0.75
Flash Scroll	79.82%	94.87%	64.91%	0.77	0.87
Flash Navigation	56.25%	72.73%	66.67%	0.70	0.71
Form Submission	56.25%	64.29%	81.82%	0.72	0.67
Navigation Path	55.56%	90.00%	56.25%	0.69	0.80
Bulk Action	75.00%	77.78%	87.50%	0.82	0.80

[7] http://www.alexa.com/topsites (last accessed Jun 8, 2016)

Accuracy determines the percentage of events correctly diagnosed by the tool, i.e. correctly captured and correctly omitted, that matched our criterion in the observed behavior of the subjects. **Recall** indicates how many events were correctly captured with respect to all that *should* have been captured according to our observations, without considering omissions (neither right nor wrong). Finally, **precision** shows the level of false positives, i.e. out of all the captured events, the proportion of correct ones. The F1 and F2 scores work as a weighted average of these measurements:

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

$$F_2 = 5 * \frac{precision * recall}{4 * precision + recall}$$

As we explained earlier, we will give more relevance to F2 since we are interested in measuring recall over precision.

6.1.3 Discussion

Some results deserve a special explanation. High recall percentages on *Click Attempt* and *Flash Scroll* mean that missing a usability event in these cases is rare, which is coherent with the capture logics with which they are implemented. They can, however, happen. Some example of missing *Click Attempts* are clicks made for text selection that were interpreted as clicks in the observations, or clicks made specifically *after* a text selection that are not captured with the current heuristics. In the case of *Flash scroll*, the threshold is speed-based and simply missed a few scrolls that were judged fast enough in the observations. Other similar thresholds based on speed that may suffer the same effect are *Tooltip Attempt*, *Navigation Path* and *Flash Navigation*. Nevertheless, these thresholds are very conservative. In the cases of the events in group 2, namely *Bulk Action*, *Search* and *Form Submission*, the main cause of missing events is failure to detect forms correctly, or actions related to them, such as submission actions in non-standard HTML GUIs.

Events can also lead to log false positives, which are consequence of misinterpreting the user's intentions, and that's why their F2 scores can decrease in comparison with their recall.

The results on F2 show relatively high scores (above .65) in the performance of USF with all the events. Nevertheless, it is possible to identify among the F2 scores two groups of usability events. The group that performed better (.75 < F2 < .97 - *Click Attempt*, *Tooltip Attempt*, *Flash Scroll*, *Navigation Path*, *Bulk Action*), matches mostly the usability events that depend on user interpretation. The group with moderate results (.67 < F2 < .75, *Search*, *Flash Navigation*, *Form Submission*) corresponds mostly with events that involve intensive reasoning on DOM elements' structures.

Even if F2 measures are acceptable for all events, improving the detection logics for those that involve intensive analysis on DOM elements will increase the coverage of detected bad

smells. Regarding the events that show higher recall but low precision (*Tooltip Attempt*, *Flash Scroll*, *Navigation Path*), the problem that must be addressed is the false positive rate, that could lead to false positive usability smells.

We consider the results to be a positive scenario, since DOM elements interpretation can be improved more easily than user interpretation, simply processing more HTML code samples.

6.1.4 Threats to Validity

There is a number of threats to this experiment's validity. Since we ran it in two different settings (multiple subjects for user intention / multiple websites for HTML structure), we considered two separate sets of threats. In the experiment that involved subjects, we considered a number of threats that we tried to mitigate. Since we had to search for events detection in finite sessions, the experiment was affected by the **fishing** threat, i.e. searching for specific results. This was reduced by designing meaningful tasks for the users, and not providing further guidance during the experiment, to avoid forcing them to run into specific interaction problems. The **external validity threats** were reduced by selecting intentionally heterogeneous volunteers, and with diverse expertise on the analyzed websites. Also, the websites were real, and while we provided the subjects with the device to run the tests, we did not remove them from their workplaces or environments. In the case of **social threats to construct validity**, which might affect the results when subjects are afraid to be evaluated or act differently, we minimized it by recruiting only volunteers who are not aware of the process that's being evaluated, since the logger tool is unnoticeable. Besides, the process of capturing events is not affected by fearful behavior in either positive or negative ways.

Regarding the experiment involving many websites, evaluated by ourselves, we mostly considered external validity threats regarding the representativeness of the sample. To tackle this issue, we selected websites from the top 50 websites on the Alexa ranking.

6.2 Evaluation of Automated Usability Smells Detection

The evaluation regarding the automated detection of usability smells assesses how the USF tool performs in real websites, both alone and in contrast with traditional user tests where an expert observes users completing tasks on a website, in a controlled environment. The method used for our assessment is the Goal–Question–Metric (GQM) (Basili et al. 1994). GQM defines goals, refines them into questions, and specifies required metrics to answers these questions.

6.2.1 Experiment Definition and Planning

Our **goal** was to find out the degree to which the USF tool can complement or replace manual methods for usability assessment. For this purpose, we measured USF's reliability, the agreement with manual user tests, and the demanded resources.

We refined our goal into three **questions**, with their corresponding metrics:

> **Q1**: How many usability problems can the USF tool find with respect to a typical usability test?
>
> **M1:** Number of problems found by each approach

M2: % of problems found by USF with respect to traditional usability tests

Q2: How much time and resources does the USF tool require compared to a typical usability test?
M3: Man-hours spent on detecting problems
M4: Required resources

Q3: How reliable are USF automated results?
M5: Number of correct results.
M6: Number of false positives.

In a similar way of the experiment described in Section 6.1, we performed a preliminary evaluation on several web applications (different from the ones in the current experiment) in order to adjust the threshold parameters of usability smell finders in order to drop the false positives ratio as low as possible. The presence of false positives is the primary threat that the USF tool faces, being a highly sensitive approach for the smells it is able to detect, so the parameters adjustment is mostly related to this aspect. We found that, in order to get consistent results, different traffic loads required different thresholds. The trial runs were carried out in 2 periods of 1 month each. The final threshold parameters are shown in Appendix A.

After the preliminary evaluation, the first part of the experiment consisted in running manual usability tests on both web applications. To use as a reference, we ran a typical manual test with 9 subjects, 4 males and 5 females with ages ranging from 25 to 62 ($\bar{x}$ 35, s^2 125.25). A well-known study by Nielsen affirms that a single test with 5 subjects can detect about 85% of the usability problems on a website[8]. According to this study, the percentage of problems found with n users is $(1-(1-L)^n)$, being L the proportion of usability problems discovered while testing a single user. A typical value for L is 31%, based on their experience. Using this formula, a single test with 9 users provides about 96% of the problems, which makes the test an effective and inexpensive comparison point.

We ran the experiment on two real web applications: an online travel agency (TA application), and a merchandising shop franchise for a first division football club (FF application), both based in Argentina. The subject selection for the TA application was based on the statistics from the Argentine Ministry of Tourism[9], while for the FF application it was based on a personal interview with the retail store manager for the same franchise. By observing the demographic information, we recruited subjects that proportionally represented the groups. We worked alongside the site owners in each case to find out the most frequent tasks their users performed in their sites. A complete list of tasks appears in Appendix A.

Subjects had to complete five typical tasks on each application, while we observed their behavior and measured the completion times. In order for the subjects to feel comfortable introducing their personal data and performing purchases in a realistic way, we set up clones of the real applications. After the tests, we obtained a list of usability issues for both applications

[8] http://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/ (last accessed Jun 16, 2016)
[9] http://desarrolloturistico.gob.ar/estadistica/evyth (last accessed Jun 16, 2016)

(to gather values for metrics M1, M2, M5 and M6) and also the time and resources spent by developers / staff (for metrics M3 and M4).

The second part of the experiment was to gather the results from USF. To do this, we created one account for each real version of the aforementioned applications at the USF tool and installed the code snippet as indicated by the tool. This allowed for measuring the time to set up (for M3), resources spent (for M4). The tool collected and analyzed interaction data for a period of 5 weeks, 3 weeks in February, 2015 and 2 additional weeks in May, 2016 (for M1, M2, M5 and M6). During these periods, we used a single instance of the USF tool on a VPS (Virtual Private Server) with 1gb of RAM running Ubuntu Server 14.04.

6.2.2 Results

Regarding metric M1, the combined total number of problems found by both approaches was 27. Out of this total, having processed 52,568 events (25,556 for the FF application and 27,012 for TT), USF found 18 issues **(66.67% of the total 27)**, while the manual test found 19 **(70.37%)**, with an intersection of 10 issues **(37.04%)**. Regarding this intersection, if we solely consider the 19 problems found by the manual tests, USF found **52.63%**.

Amongst the problems that both methods found, the registration form in the TA application didn't highlight the mandatory fields, which we observed during the manual tests, and USF reported as a *Late Validation* smell. In the FF application, users wanted to return to previous steps during the checkout process, but the steps indicators were read-only; this was found in the manual tests and also reported by USF as an *Unresponsive Element* smell.

Regarding the problems exclusively found by the manual tests (i.e., undetected by the tool) we found specific application behavior or inconveniences detected by body language, which are very difficult to capture, or cannot be captured automatically at all. For example, in the FF application, users were confused by the "Add to cart" action, that did not show clearly enough that the product had been added.

Meanwhile, the problems exclusively found by the automated tool (i.e., undetected in the manual tests) were primarily due to interactions that were not covered by the tasks on the manual tests. For example, in the FF application, the tool detected that many users clicked on a read-only welcome banner that was not mentioned in the tasks, probably expecting to navigate to another section.

Concerning M5 and M6 for the USF results, the total number of problems found was actually higher than the 18 valid ones. The tool found 6 extra smells that were false positives, which amounts to 75% of true positives (M5) and 25% of false positives (M6). As an example of false positive, a *Late Validation* was found in the FF application for the "Add to cart" button, which was actually a submit button for a form. After submission, the same form reappeared, since adding a product to the cart does not navigate away from the product's page. This was interpreted by the events capturer as a failed submission, when in fact the product was being added to the cart. Table 4 shows the detected smells separated by kind, along with the True/False Positive count.

Table 4. Results of the experiment on usability smells capture

	True Positives	False Positives	Total
Scarce Search Results	1	1	2
Late Validation	3	2	5
Unresponsive Element	4	1	5
Free Input For Limited Values	3	0	3
Misleading Link	1	1	2
Short Input	2	0	2
Undescriptive Element	0	1	1
No Processing Page	3	0	3
Abandoned Form	1	0	1
Total	18	6	24

Regarding man-hours (metric M3), the total time consumed by the manual tests was **5h 40' for TA**, and **6h 20' for FF** (for tasks' design, setup of clone applications, user tests, and analyses – excluding travel times), while the USF approach required **45' for FF** and **35' for TA** (for tool's setup and results' inspection). Regarding resources (metric M4), the manual test required 9 subjects for each application, two usability experts to run the tests, and a web developer to set up the applications' clones on a separate server. USF required a web developer capable of adding the code snippet to the evaluated application.

6.2.3 Discussion

The first question we attempted to answer with our experiment's definition was Q1: "*How many problems can the USF tool find with respect to a typical usability test?*". The experiment showed that the USF tool found half of the problems that the manual tests had found, which we consider a positive result, since it means that USF is able to find similar problems than a proven technique does – albeit in an automated way. However, some of the problems detected only on the manual tests (4/27) were too application-specific, and required human reasoning, which means USF will probably never be able to find them. Nevertheless, USF did find problems that the manual test did not. This happened for two reasons: (1) the tasks set a fixed course of actions for the subjects, which narrows down the possible usability issues they will run into, and (2) the limit of 9 users (and so the alleged 96% coverage) also restricts the number of problems that the expert can find. USF, being based on real interactions from a larger mass of users, was able to find other results.

Regarding Q3: "*How accurate are USF automated results?*", even if the false positives ratio was noticeable (25%), and leaves room for improvement, we believe it does not make the approach / tool unreliable. Also, the failed results can be fixed in most cases by improving the heuristics.

Finally, we believe that Q2: "*How much time and resources does the USF tool require compared to a typical usability test?*" shows a strong point in favor of USF. Even if it does require time to show substantial results, depending on the site's traffic, the cost and effort is

minimum, so there is a big cost/benefit ratio for those who cannot spare resources, or lack the knowledge for running a traditional usability test.

By answering these questions, we met our goal, i.e., we got a first perspective on the extent to which our approach could replace manual usability testing. The answer is two-fold. Firstly, the results indicate that the tool can find meaningful smells, and comparable to the problems found by manual testing, although it cannot replace human reasoning required for some of them. Secondly, the cost of analyzing a web application is extremely low, compared to manual testing. Thus, the tool can be (1) an alternative with nearly zero-cost for developers with no resources to run usability tests, and also (2) a good complement for manual testing, since it may find problems that manual testing does not cover.

Another positive aspect of the experiment was the opportunity to add new usability smells to our tool, since the manual tests provided new insights. For example, a remarkable issue found by the manual test in the TA application was a case of **banner blindness**: users had to visit a specific section, but since the link looked like an advertisement (though it was not) nobody found it. This behavior can be generalized and detected as a usability smell.

6.2.4 Threats to Validity

Our experiment was naturally subject to potential threats that required our attention. The most relevant **internal validity** threats were dealt with from the experiment's design. To mitigate **instrumentation threats** regarding the users' environment, we allowed them to use their own devices, in their own homes or workplaces. Regarding **external validity** threats, we looked for user representativeness on each application, as earlier described. Moreover, we faced the **fishing threat to conclusion validity**, that could threat the experiment in two possible ways: by running the manual experiment knowing the USF results, and vice versa, i.e., by knowing the manual experiment's results before adjusting the USF tool. For the first case, we didn't inspect the results from USF until we had finished manual testing, to prevent the latter results from being biased by our knowledge on the tool's performance. We also reduced this threat by designing common, representative tasks for each website, without providing specific instructions other than the tasks themselves. For the reverse case of phishing threat, we set the parameters for the usability smells finders based on the trial runs on different websites than the ones involved in the experiment. Regarding **construction validity**, we minimized evaluation apprehension by recruiting only volunteers.

7 Concluding Remarks and Further Work

The main contribution of the work presented in this article are summarized below:

- a process for the systematic detection and correction of usability smells that appear in user events on highly interactive websites; most importantly, these usability smells that we propose are specific enough to suggest solutions in terms of refactorings.
- a tool, based on that process, that discovers 16 usability smells automatically and reports them along with suggested refactorings to evaluators, who do not need to be usability experts;

- a framework with a client-side and a server-side component that allows extensibility at both ends, i.e., incorporating new events to be captured on the client, and adding new usability smells, so the detection mechanism can improve over time, or may be adapted to different contexts (like mobile applications, or specific domains);
- an empirical validation for the process applied in real web applications; through these validations we proved that our process can find a number of meaningful usability smells with almost no setup cost.

The approach has however some limitations. There are some usability problems that require human reasoning, so the automated solution naturally cannot detect them. Also, the number of usability smells we are able to detect is restricted to those that users repeatedly run into. Consequently, our future work includes adding to our tool the possibility of automatically finding problems through other techniques such as statically analyzing the DOM structure of each webpage. Another limitation to our process is that it doesn't have training information on how tasks are supposed to be done, nor it gathers explicit feedback from end-users. Nevertheless, we purposely decided to bear with this limitation from the start, since we strive for a fully automated and unattended process that requires the least possible effort at setup.

We have assessed the performance of both usability events and usability smells detection with separate experiments. In the Smells Experiment, the automated tool was able to find 52% of usability smells manually found in the control applications, while the Events Experiment showed a high level of accordance with respect to the manual analysis, indicated by an average 0.75 F2 score. Despite the numeric results, by running the experiment we were able to spot the weak parts of our detection process, and we are already working on improving them.

There are also other lines of work we are pursuing within the presented approach. First, since the number of usability events can grow very quickly in deployed applications, we plan to apply data mining techniques to select representative samples logging a portion of the continuous flow of events without sacrificing detection power. This can be done with different criteria:

- Events kind / Affected elements: we might choose to log only a certain amount or percentage of events on a specific smell or affected DOM element.
- User demographic data / Device: having collected information on the location or device used of our visitors, we could select a sample depending on that data, and log events proportionally.
- Time of event: we might need to detect issues that happen at all times in equal distribution, or restrict to a specific time of day.

Moreover, the experiments showed that for a few usability smells, the best default threshold values were dependent on the traffic load, so we set them manually for each case. Future work involves studying how these values relate to the traffic in order to set them automatically.

We are also extending the approach to automate the refactoring implementation. For this purpose we will integrate our previous work on client-side refactorings, which implement most catalogued usability refactorings (Garrido, Firmenich, et al. 2013). This integration will permit a fully automated process where not only we can automate the detection of bad smells, but also the refactoring themselves.

Finally, we are investigating stronger ways of protecting users' privacy by anonymization techniques. With a sound work on this area, we could even start logging full user sessions without compromising their privacy and potentially obtaining better results in the detection of usability smells.

References

Ambler, S. & Sadalage, P., 2006. *Refactoring databases: Evolutionary database design*, Addison Wesley.

Apaolaza, A., Harper, S. & Jay, C., 2015. Longitudinal Analysis of Low-Level Web Interaction through Micro Behaviours. In *Proceedings of the 26th ACM Conference on Hypertext & Social Media - HT '15*. New York, New York, USA: ACM Press, pp. 337–340.

Atterer, R., Wnuk, M. & Schmidt, A., 2006. Knowing the user's every move: user activity tracking for website usability evaluation and implicit interaction. In *Proceedings of the 15th Int. Conf. on World Wide Web*. pp. 203–212.

Basili, V.R., Caldiera, G. & Rombach, H.D., 1994. The goal question metric approach. *Encyclopedia of Software Engineering*, 1, pp.528–532.

Breslav, S., Khan, A. & Hornbæk, K., 2014. Mimic. In *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces - AVI '14*. New York, New York, USA: ACM Press, pp. 245–252.

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

Cabot, J. & Gómez, C., 2008. A catalogue of refactorings for navigation models. In *Proceedings - 8th International Conference on Web Engineering, ICWE 2008*. pp. 75–85.

Carta, T., Paternò, F. & Santana, V. de, 2011. Web usability probe: a tool for supporting remote usability evaluation of web sites. *Human-Computer Interaction–INTERACT 2011*, pp.349–357.

Chi, E.H., 2002. Improving Web Usability Through Visualization. *IEEE Internet Computing*, 6(2), pp.64–71.

Dig, D., 2011. A refactoring approach to parallelism. *IEEE Software*, 28(1), pp.17–22.

Distante, D. et al., 2014. Business processes refactoring to improve usability in E-commerce applications. *Electronic Commerce Research*, 14(4), pp.497–529.

van Duyne, D.K., Landay, J. a. & Hong, J.I., 2006. The Design of Sites.

Fernandez, A., Insfran, E. & Abrahão, S., 2011. Usability evaluation methods for the web: A systematic mapping study. *Information and Software Technology*, 53(8), pp.789–817.

Fowler, M., 1999. *Refactoring: improving the design of existing code*, Addison-Wesley.

Garrido, A., Rossi, G., et al., 2013. Improving accessibility of Web interfaces: refactoring to the rescue. *Universal Access in the Information Society*, pp.1–13.

Garrido, A., Firmenich, S., et al., 2013. Personalized web accessibility using client-side refactoring. *IEEE Internet Computing*, 17(4), pp.58–66.

Garrido, A., Rossi, G. & Distante, D., 2007. Model refactoring in web applications. In *Proceedings - 9th IEEE International Symposium on Web Site Evolution, WSE 2007*. pp. 89–96.

Garrido, A., Rossi, G. & Distante, D., 2011. Refactoring for Usability in Web Applications. *IEEE Software*, 28(3), pp.60–67.

Gregg, D.G. & Walczak, S., 2010. The relationship between website quality, trust and price premiums at online auctions. *Electronic Commerce Research*, 10(1), pp.1–25.

Grigera, J. et al., 2016. Assessing refactorings for usability in e-commerce applications. *Empirical Software Engineering*, 21(3), pp.1224–1271.

Grigera, J., Garrido, A. & Rivero, J.M., 2014. A Tool for Detecting Bad Usability Smells in an Automatic Way. In *Proceedings of the 14th International Conference in Web Engineering (ICWE 2014)*. Springer, pp. 490–493.

Harms, P. & Grabowski, J., 2014. Usage-based Automatic Detection of Usability Smells. In *Proceedings of Human-Centered Software Engineering*. pp. 217–234.

Harold, E.R., 2008. *Refactoring HTML: Improving the Design of Existing Web Applications*, Addison-Wesley.

Hilbert, D.M. & Redmiles, D.F., 2000. Extracting usability information from user interface events. *ACM Computing Surveys*, 32(4), pp.384–421.

Hong, J.I. et al., 2001. WebQuilt: A proxy-based approach to remote web usability testing. *ACM Transactions on Information Systems*, 19(3), pp.263–285.

Hornbæk, K., 2006. Current practice in measuring usability: Challenges to usability studies and research. *International Journal of Human Computer Studies*, 64(2), pp.79–102.

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

Ivory, M.Y. & Hearst, M. a., 2001. The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys*, 33(4), pp.470–516.

Lanza, M. & Marinescu, R., 2006. *Object-oriented metrics in practice*, Springer.

Levenshtein, V., 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet physics doklady*, 10, pp.707–710.

Nah, F.F.-H., 2004. A study on tolerable waiting time: how long are Web users willing to wait? *Behaviour & Information Technology*, 23(3), pp.153–163.

Nebeling, M., Speicher, M. & Norrie, M., 2013a. CrowdStudy: General toolkit for crowdsourced evaluation of web interfaces. *EICS 2013 - Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pp.255–264.

Nebeling, M., Speicher, M. & Norrie, M., 2013b. W3touch: Metrics-based Web Page Adaptation for Touch. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13*, p.2311.

Nielsen, J. & Loranger, H., 2006. *Prioritizing Web Usability* C. Peri, ed., Pearson Education.

Okada, H. & Fujioka, R., 2008. Automated Methods for Webpage Usability & Accessibility Evaluations. In S. Pinder, ed. *Advances in Human Computer Interaction*. InTech, pp. 351–364.

Opdyke, W., 1992. *Refactoring Object-Oriented Frameworks*. Univ. of Illinois at Urbana-Champaign.

Paternò, F., Schiavone, A.G. & Pitardi, P., 2016. Timelines for Mobile Web Usability Evaluation. In *Proceedings of the International Working Conference on Advanced Visual Interfaces - AVI '16*. New York, New York, USA: ACM Press, pp. 88–91.

Reis, D.C. et al., 2004. Automatic web news extraction using tree edit distance. *Proceedings of the 13th conference on World Wide Web WWW 04*, p.502.

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

Santana, V.F. de & 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, pp.40–49.

Seckler, M. et al., 2014. Designing usable web forms. *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI '14*, pp.1275–1284.

Seffah, A. et al., 2006. Usability measurement and metrics: A consolidated model. *Software Quality Journal*, 14(2), pp.159–178.

Speicher, M., Both, A. & Gaedke, M., 2015. S.O.S.: Does Your Search Engine Results Page (SERP) Need Help? In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15*. New York: ACM Press, pp. 1005–1014.

Wroblewski, L., 2008. Web Form Design: Filling in the Blanks. *Interactions*, 0(October), p.226.

APPENDIX A: Experiments' Details

Events Experiment – Usability Events (Section 6.1)

Group 1 – Heuristics involving user intention:

Users' experience on the tested websites, from 1 (no experience) to 5 (frequent user).

	mercadolibre.com	info.unlp.edu.ar	stackoverflow.com	Gender	Age
Volunteer 1			3	M	34
Volunteer 2			3	F	36
Volunteer 3	2	1		M	34
Volunteer 4	5	1		F	33
Volunteer 5	1	1		F	64
Volunteer 6	4		5	F	27
Volunteer 7		3	3	M	40
Volunteer 8	1	2		F	37
Volunteer 9	3	5	2	M	35
Volunteer 10	3	1		M	35
Volunteer 11	4	1		M	39
Volunteer 12	4		3	M	32
Volunteer 13		3	3	M	32
Volunteer 14		3	3	F	27
Volunteer 15	3	4		F	30

Tasks

Site 1: mercadolibre.com

1. Search for a used Android tablet to buy and get into its details.
2. Search for flaws on the tablet.
3. Check for stock on the tablet.
4. Find out the tablet's price.
5. Find out what the blue icons are for.
6. Register as a user.
7. Find out how "MercadoEnvios" works, and who pays for the shipping.
   (N: "MercadoEnvios" is a shipping service offered by the website).
8. Find out what's the percentage of the total cost that the site gets out of a sold item.

Site 2: info.unlp.edu.ar

1. Find out what are the first 2 icons on the upper bar for.
2. Search for an article on predictive technology. Find out what the user on the photo is browsing on his phone.
3. Go back to the homepage.
4. Search for the studies plan for "Licenciado en Sistemas". Find out whether it requires a thesis at the end of the plan.
5. Find out who the dean of the faculty is.

Site 3: stackoverflow.com

1. Find out what the icons at "How questions" represent.
2. Find out how can one add a CSS class to an element using JQuery. Vote for the best answer.
3. Search for the top JQuery questions. See what's the most recent unanswered one.
4. Register to the site.

Group 2 – Heuristics involving DOM elements detection

Evaluated sites:

• mercadolibre.com • info.unlp.edu.ar • stackoverflow.com • imgur.com • groups.google.com • linkedin.com • paypal.com • mail.yahoo.com • amazon.com • aws.amazon.com • duolingo.com	• mail.live.com • instagram.com • aliexpress.com • clarin.com • lanacion.com.ar • infobae.com • ole.com.ar • tn.com.ar • despegar.com.ar • taringa.net • perfil.com

Smells Experiment – Usability Smells Detection Evaluation (Section 6.2)

Subjects info

	Weekly hours browsing	E-commerce usage level	Gender	Age
Volunteer 1	5	3	M	34
Volunteer 2	12	1	F	36
Volunteer 3	15	4	F	62
Volunteer 4	25	4	M	25
Volunteer 5	20	5	M	29
Volunteer 6	20	1	F	37
Volunteer 7	20	1	M	24
Volunteer 8	18	3	F	36
Volunteer 9	25	3	M	32

Tasks

Site 1: TA Application

1. Search for a travel package to a given destination. Find the cheapest one.
2. Contact the agency for more information on the packages.
3. Search for a flight to a given destination.

4. Get prices for traveler assistance for a 14-day trip to Europe.
5. Subscribe to the newsletter.

Site 2: FF Application

1. Find out the cost of the team's official football shirt.
2. Check the shirt's sizes and select the best fit. Add a shirt to the cart and go back home.
3. Find out if the shop sells backpacks. If they do, add one to the cart.
4. Complete the checkout process.
5. Register to the site. (*N. it is a step of the checkout process for unsigned users*)

<u>Threshold values for individual usability smell finders</u>

In the cases were thresholds were not bound to proportions (for example, the finder for *Unresponsive Element*) we used the traffic load as a reference.

The minimum event count for all smells was set to 50.

Smell Id	Usability Smell	Thresholds
1	Undescriptive Element	30 Tooltip Attempt in a 5-day window
2	Misleading Link	12 Flash Navigation in a 5-day window
3	No Processing Page	10000 ms average waiting time
4	Free Input For Limited Values	0.2 min group size rate 0.1 max rate of unique values Levenshtein edit distance < 4
5	Unformatted Input	0.7 min formatted values rate
6	Short Input	0.3 long input values rate
7	Unnecessary Bulk Action	0.8 min single operation rate
8	Overlooked Content	0.4 min events in same y-area rate 100 pixels tolerance for y-area
9	Distant Content	20 Navigation Paths in a 5-day window
10	No Client Validation	0.6 reject rate
11	Late Validation	0.6 reject rate
12	Abandoned Form	0.4 drop-off rate
13	Scarce Search Results	0.6 no-results rate
14	Useless Search Results	0.4 unfollowed-results rate
15	Wrong Default Value	0.8 non-default selection rate
16	Unresponsive Element	12 Click Attempts in a 5-day window

APPENDIX B: Detection Logics for Usability Smells

Undescriptive Element

- Description: Many users try to get a tooltip from the element. This may be due to the element not being self-descriptive enough.
- Associated Usability Event: Tooltip Attempt.
- Detection Process: for each element affected with Tooltip Attempts, count all events inside the specified time window. If the count exceeds the minimum threshold, the smell is detected
- Adjustable Parameters:
  - Minimum Events: *how many tooltip attempts trigger the smell?*
  - Time window: *in what timespan are the events counted? E.g. "one week back from today".*
- Recommended Refactorings
  - Rename Element
  - Change Widget

Misleading Link

- Description: Users frequently visit the linked contents, only to return shortly after.
- Associated Usability Event: Tooltip Attempt / Flash Navigation.
- Detection Process: for each element affected with Tooltip Attempts or Flash Navigations, count all events inside the specified time window. If the count exceeds the minimum threshold, the smell is detected
- Adjustable Parameters:
  - Tooltip Attempt Minimum Events: *how many tooltip attempts are considered before triggering the smell?*
  - Flash Navigation Attempt Minimum Events: *how many flash navigations are considered before triggering the smell?*
  - Time window: *in what timespan are the events counted? E.g. "one week back from today".*
- Recommended Refactorings: Rename Anchor

No Processing Page

- Description: A page usually takes long to process, and no information is presented to the user to indicate there is an ongoing process.
- Associated Usability Event: Long Request.
- Detection Process: for each page affected with Long Request events, get the average waiting time, if this time exceeds a time threshold, the smell is detected.
- Adjustable Parameters:
  - Average Waiting Time: *what is the average tolerated time (ms) before triggering the smell?*
- Recommended Refactoring: Add Processing Page

Unformatted Input

- Description: A simple text input asks users to fill in data in a specific format, instead of offering

a more adequate widget.

- Associated Usability Event: Text Input
- Detection Process
  1. For each text input affected by Text Input events, collect all values.
  2. Run the regular expressions that represent the preformatted values (e.g. phones) by the values and count the amount of matches.
  3. If the amount of matched values exceeds a proportion threshold for a particular regular expression, trigger the smell.
- Adjustable Parameters:
  - Formated Values Proportion: *what percentage of values must match a particular format before triggering the smell?*
  - Regular Expressions: *what formats should we look for in the values?*
- Recommended Refactorings
  - Change Widget
  - Add an Assistance Activity

Short Input

- Description: An input box is shorter than the values usually typed on it, often hiding the full typed text.
- Associated Usability Event: Text Input
- Detection Process:
  1. For each text input affected by Text Input events, collect all events.
  2. Calculate the proportion of events with exceeded text (text values larger in width than the text box's width, calculated from the capital "M" width of the used font).
- Adjustable Parameters
  - Exceeded Values Proportion: *how many values must exceed the text input width before triggering the smell?*
- Recommended Refactoring: Resize Element

Unnecessary Bulk Actions

- Description: a form contains a list of items, and each one has a checkbox. Users can perform different actions over a subset of those items by first selecting some of them using those checkboxes, and then selecting an action. However most of the time users only apply the actions on a single item at a time rather than many, rendering these mechanics unnecessarily complicated.
- Associated Usability Event: Bulk Action.
- Detection Process:
  3. For each form affected with events, take all Bulk Action threats related to it.
  4. For a given form, group the events by the action applied (e.g. "delete").
  5. For each group related to an action, split the events in two sets: those where the action was applied on a single item, and those where the action applied to many.

6. If the set of events with a single item is proportionally larger than the other set by a given threshold, then the bad smell is detected for that action.

- Adjustable Parameters:
  - Minimum Single Operation Proportion: *what is the minimum proportion of threats with a single item to consider the presence of the bad smell?*
- Recommended Refactorings:
  - D*istribute Menu*
  - *Replace widget*
  - *Add Contextual Menu.*

Overlooked Content

- Description: Users scroll to an area too quickly. It is possible that they don't read the contents between the top of the page and said area
- Associated Usability Event: Flash Scrolling
- Detection Process
  - For each page affected with Flash Scrolling, collect the events.
  - Split the events by scrolling direction.
  - For each scrolling direction, group by landing y-position proximity.
  - For each group, calculate the proportion of Flash Scrolling events with respect to the total events count.
  - If a group represents a proportion of the total events larger than a given threshold, then the smell is triggered.
- Adjustable Parameters
  - Min Group Proportion: *what is the minimum proportion for a given y-area landing group over all the events that trigger the smell?*
  - Landing Position Tolerance: *how close should 2 given Flash Scrollings of the same direction land to be considered in the same group?*
- Recommended Refactorings
  - Split Page
  - Remove Redundant Content

Distant Content

- Description: Users often follow a path of many navigation steps too quickly to get to a point. They do not seem to stop in the middle nodes and see/interact with their contents.
- Associated Usability Event: Navigation Path
- Detection Process:
  - Group together all Navigation Paths with the same navigation nodes in the same order.
  - For each group, count all events inside the specified time window. If the count exceeds the minimum threshold, the smell is detected

- Adjustable Parameters:
  - Navigation Path Minimum Events: *how many equivalent navigations are considered before triggering the smell?*
  - Time window: *in what timespan are the events counted? E.g. "one week back from today".*
- Recommended Refactoring: Add Link

No Client Validation

- Description: A form lets users submit wrong data and only informs the issues after the submission.
- Associated Usability Event: Form Submission
- Detection Process: For each affected form, count all Submission events and calculate the proportion of rejections after a navigation (server-side validation). If this proportion exceeds a threshold, the smell is detected.
- Adjustable Parameters:
  - Reject proportion: *how many rejects over the total submissions trigger a smell?*
- Recommended Refactorings: Anticipate Validation

Late Validation

- Description: A form lets users submit wrong data and only informs the issues after they click the submit button.
- Associated Usability Event: Form Submission
- Detection Process: For each affected form, count all Submission events and calculate the proportion of rejections after submission, but no navigation ensues (client-side validation). If this proportion exceeds a threshold, the smell is detected.
- Adjustable Parameters:
  - Reject proportion: *how many rejects over the total submissions trigger a smell?*
- Recommended Refactoring: Anticipate Validation

Abandoned Form

- Description: A form is usually abandoned by the users before being submitted.
- Associated Usability Events: Form Submission / Abandoned Submission
- Detection Process: For each affected form, calculate the proportion of abandonments vs. submissions (either failed or successful). If the drop-off rate exceeds a threshold, the smell is detected.
- Adjustable Parameters:
  - Drop-Off Proportion: *how many abandonments over submissions trigger a smell for a given form?*
- Recommended Refactorings
  - Split Activity
  - Postpone activity

Useless Search Results

- Description: A search form's results are rarely followed by users.
- Associated Usability Event: Search
- Detection Process:
  - For each search form, gather all Search events that brought results.
  - Calculate the proportion of searches that follow a navigation from one of the results.
  - If the proportion exceeds a threshold, the smell is detected.
- Adjustable Parameters:
  - Unfollowed Results Proportion: *how many successful searches, in proportion, were not followed by a navigation from one of the results?*
- Recommended Refactoring: Add Autocomplete

Wrong Default Value

- Description: A list of values is presented to the user either in a select box or a list of radio buttons. A particular value is selected too often but users must explicitly select it anyways.
- Associated Usability Event: Option Selection
- Detection Process: For each affected group of options, compare the ones with the default selection with the ones that include a different selection. If the latter are more than the former by a given margin, the smell is detected.
- Adjustable Parameters:
  - Non-Default Selection Proportion: *how many selections of non-default values with respect to the total should trigger a smell?*
- Recommended Refactoring: Set Default Value

Unresponsive Element

- Description: An element is often clicked upon, but it does not trigger any action.
- Associated Usability Event: Click Attempt
- Detection Process: for each element affected with Click Attempts, count all events inside the specified time window. If the count exceeds the minimum threshold, the smell is detected
- Adjustable Parameters:
  - Minimum Events: *how many click attempts trigger the smell?*
  - Time window: *in what timespan are the events counted? E.g. "one week back from today".*
- Recommended Refactorings
  - Turn Attribute Into Link
  - Change Widget

Figure Captions

Fig. 1. Screenshot of the heading of Google Drive in May'2016

Fig. 2. The three steps of the process

Fig. 3. Usability Smells Detection

Fig. 4. Sample data for detection of Free Input for Limited Values

Fig. 5.a. Screenshot of a web application under analysis

Fig. 5.b. Screenshots of the tool displaying captured usability smells

Fig. 5.c. Screenshot of bad smell report