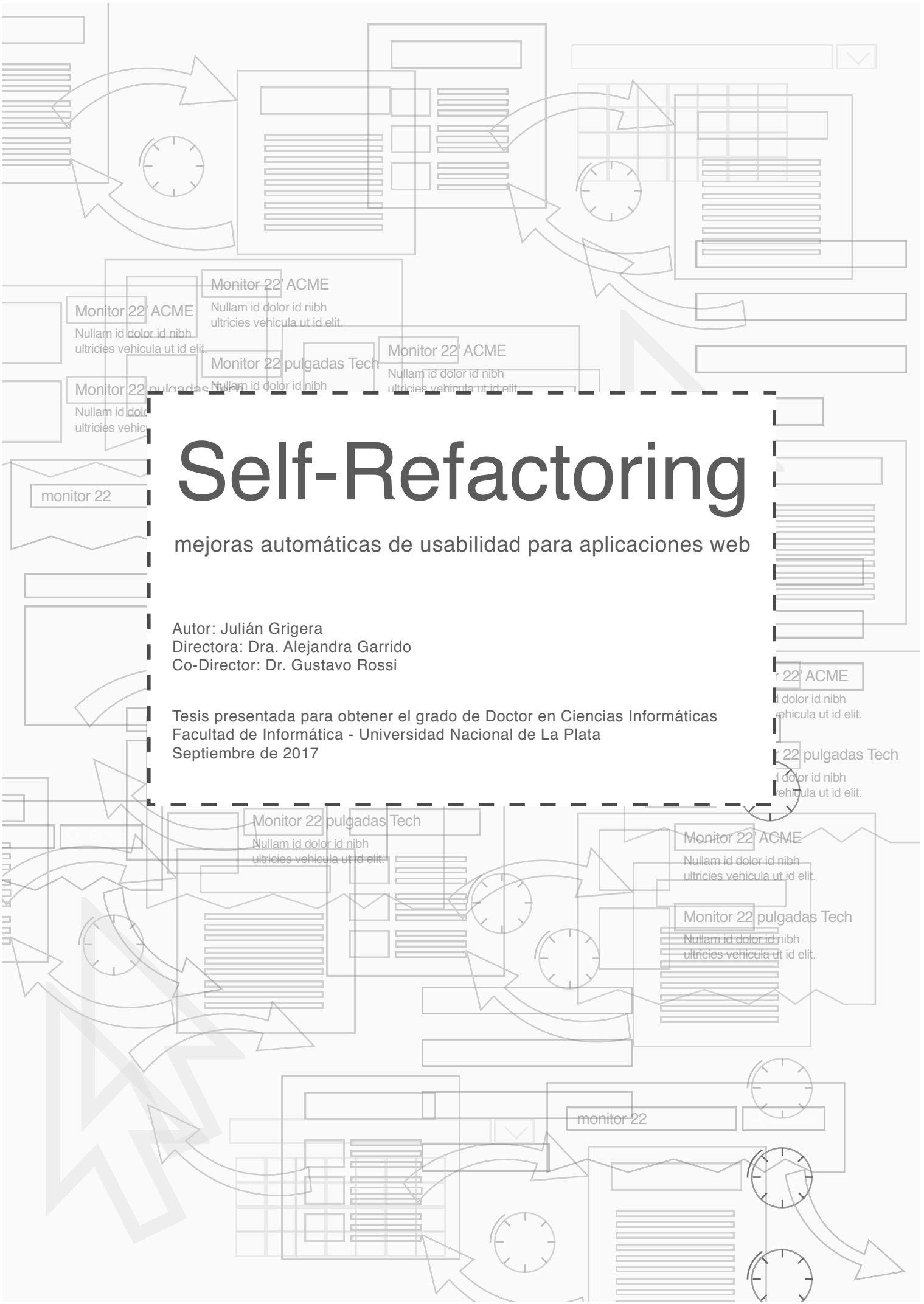

Self-Refactoring

mejoras automáticas de usabilidad para aplicaciones web

Autor: Julián Grigera
Directora: Dra. Alejandra Garrido
Co-Director: Dr. Gustavo Rossi

Tesis presentada para obtener el grado de Doctor en Ciencias Informáticas
Facultad de Informática - Universidad Nacional de La Plata
Septiembre de 2017

Índice

Agradecimientos

“Quiero agradecer a la academia…”

Este trabajo sólo puede existir con la dirección imposible de Alejandra y Gustavo. No existe una directora con el empuje y la dedicación de Alejandra, con el compromiso absoluto desde el día 0 hasta hoy. Tampoco un director con la trayectoria y generosidad de Gustavo. Gracias a ellos y su invaluable guía.

También quiero agradecer a todos mis colegas en LIFIA. Particularmente a Matías Rivero, que tuvo mucho que ver con el origen de este trabajo, y a Martín Zanotti. A Diego y a Sergio por toda su ayuda durante este tiempo.

Agradezco a mi familia, creo que hasta el día de hoy no terminan de entender qué es lo que hago, lo que hace su apoyo aún más valioso. Y a Vicky, no puedo empezar a explicar lo importante que es para mí. Pero se dan una idea. Todo esto está dedicado a ella.

Publicaciones

Alejandra Garrido, Sergio Firmenich, Gustavo Rossi, Julián Grigera, Nuria Medina-Medina, Ivana Harari: **Personalized web accessibility using client-side refactoring**.
IEEE Internet Computing, 17(4), 58-66 (2013).

Alejandra Garrido, Gustavo Rossi, Nuria Medina-Medina, Julián Grigera, Sergio Firmenich:
Improving accessibility of Web interfaces: refactoring to the rescue.
Universal Access in the Information Society 13(4): 387-399 (2014)

Julián Grigera, Alejandra Garrido, José Matías Rivero:
A Tool for Detecting Bad Usability Smells in an Automatic Way.
International Conference of Web Engineering 2014: 490-493

Julián Grigera, Alejandra Garrido, José Ignacio Panach, Damiano Distante, Gustavo Rossi:
Assessing refactorings for usability in e-commerce applications.
Empirical Software Engineering 21(3): 1224-1271 (2016)

Julián Grigera, Alejandra Garrido, José Matías Rivero, Gustavo Rossi:
Automatic detection of usability smells in web applications.
International Journal of Human-Computer Studies. 97: 129-148 (2017)

Julián Grigera, Alejandra Garrido, Gustavo Rossi:
Kobold: Web Usability as a Service.
International Conference on Automated Software Engineering: 969-974 (2017). ACM.
(aceptado para publicación)

Alejandra Garrido, Sergio Firmenich, Julián Grigera, Gustavo Rossi:
Data-Driven Usability Refactoring: Tools and Challenges.
ASE Workshops 2017 - 2017 6th IEEE/ACM International Workshop on Software Mining (SoftwareMining 2017) (aceptado para publicación)

Sergio Firmenich, Alejandra Garrido, Julián Grigera, José Matias Rivero, Gustavo Rossi:
Usability Improvement through A/B Testing and Refactoring.
Software Quality Journal (SQJO) (en revisión)

Resumen

La usabilidad en las aplicaciones web es un aspecto fundamental, pero en muchos casos relegado por diferentes motivos como la falta de personal experimentado, o los altos costos. Si bien las grandes compañías suelen estar preparadas para dedicar los recursos necesarios a mejorar la usabilidad de sus aplicaciones, las pequeñas y medianas suelen utilizarlos en otros aspectos. Para ayudar a bajar estos costos, han surgido herramientas que definición y ejecución remota de pruebas de usabilidad, o recolección de estadísticas de forma automatizada, pero igualmente se requiere de expertos que diseñen las pruebas, interpreten los reportes o visualizaciones en busca de problemas, y diseñen soluciones a los mismos, que los desarrolladores deberán implementar.

En este trabajo se propone un enfoque para hallar problemas de usabilidad automáticamente en aplicaciones web, basados en el análisis de eventos de interacción de usuarios finales. Para cada uno de estos problemas de usabilidad encontrados, existe además una solución que puede sugerirse para resolver el problema. En algunos casos, es incluso posible aplicar estas soluciones automáticamente. En este enfoque, los problemas de usabilidad se definen como “usability smells” y las soluciones como “usability refactorings”, ambos términos adaptados de la jerga del refactoring de código. Los usability smells, en este contexto, son problemas que afectan la interacción por parte de los usuarios finales, mientras que los usability refactorings son transformaciones que aplican soluciones documentadas para resolver esos problemas.

Como prueba de concepto se implementó Kobold: una herramienta capaz de realizar todo lo que se propone en este trabajo. La herramienta funciona como un servicio (SaaS – *Software as a Service*), y no requiere de casi ningún esfuerzo de instalación. Al incorporar Kobold en una aplicación web, se comienza a capturar la interacción de los usuarios, y los reportes de problemas se muestran apenas un número suficiente de usuarios se topa con los mismos. Como los usability smells son problemas bien descritos, pueden ser interpretados por cualquier desarrollador, aunque no tenga experiencia en usabilidad. De la misma forma, los refactorings que se sugieren como solución pueden ser aplicados automáticamente y en producción, gracias a la implementación de refactorings del lado del cliente, que permiten alterar la aplicación sin modificar su código.

De esta manera, Kobold se presenta como una herramienta que puede resultar de utilidad tanto para desarrolladores como para expertos en usabilidad. En resumen, lo que se quiere obtener con Kobold es, como mínimo, una herramienta confiable que con un mínimo esfuerzo de configuración pueda rápidamente comenzar a brindar asesoramiento sobre usabilidad en aplicaciones que ya se encuentran corriendo en producción, y que pueda ser configurada para detectar diferentes tipos de problemas. La audiencia para esta herramienta sería de desarrolladores con experiencia en usabilidad, que quisieran tener un panorama rápido de las interacciones reales que realiza la masa de usuario, y probablemente reparar rápidamente algunos de estos problemas del lado del cliente. Más aun, esto allanaría el camino para conseguir un objetivo más ambicioso: un mecanismo confiable que permita la auto-reparación de aplicaciones web, que incluso los desarrolladores sin experiencia en usabilidad puedan utilizar para corregir los usability smells en sus aplicaciones.

El trabajo presentado incluye validaciones empíricas que comprueban la factibilidad del enfoque y su implementación en todas las etapas: captura de eventos de interacción, detección de usability smells y aplicación de usability refactorings.

Capítulo 1. Introducción

Las aplicaciones web nos asisten en muchas de nuestras actividades diarias, como la compra de productos, leer noticias, interactuar socialmente, administrar nuestras cuentas bancarias, planificar viajes o pedir turno con un médico. Cada día surgen nuevas aplicaciones, ampliando nuestras posibilidades de realizar tareas cómodamente desde nuestros hogares y, aun así, muchas veces estas aplicaciones sufren de problemas de usabilidad que las vuelven demasiado complicadas de utilizar, tornándoles frustrantes para los usuarios. Según Jakob Nielsen, conocida figura en el campo de la usabilidad, *la usabilidad domina la web* (*usability rules the web*) y es crucial para el éxito de cualquier aplicación (Nielsen and Loranger, 2006; Gregg and Walczak, 2010). Las compañías reconocen que la competencia es tan alta que difícilmente puedan sobrevivir de no invertir en usabilidad, aunque sigue siendo muy costoso a pesar del progreso que se ha dado últimamente en el área, y en las herramientas para aplicarla.

Uno de los métodos más populares para evaluar usabilidad es mediante los tests de usabilidad, en particular los tests de usuario (Rubin and Chisnell, 2008). El beneficio de estos tests radica en que, en contraste con los métodos de inspección (como la evaluación de heurísticas), capturan datos de uso sobre experiencias reales de usuario, o al menos realistas. La desventaja, sin embargo, yace en que los tests de usuario requieren reclutar voluntarios y emplear mucho tiempo y recursos, como expertos en usabilidad que deben diseñar los tests y tareas, y luego conducir las sesiones de prueba para luego evaluar los resultados y elaborar un diagnóstico. Además de esto, hace falta hallar e implementar soluciones para los problemas detectados.

Para ahorrar en costo, tiempo, y recursos que involucran los tests de usuario, han surgido diferentes alternativas para realizar estos tests de forma remota e incluso automatizada en algunas de sus etapas. Existen entonces diferentes métodos para capturar la interacción de los usuarios en forma de eventos, y luego realizar algún tipo de análisis para ayudar a los expertos a descubrir patrones de uso (Santana and Baranauskas, 2015). Los resultados suelen ser presentados mediante sofisticadas técnicas de visualización que permiten comparar secuencias reales de eventos con otras secuencias óptimas de interacción para descubrir desvíos o diferencias. Sin embargo, es inusual que estas herramientas provean sugerencias que ayuden a los diseñadores mejorar sus interfaces; siempre hace falta un experto que sea capaz de dilucidar problemas de usabilidad a partir de estos desvíos en las visualizaciones, y eventualmente hallar diferentes soluciones para los dichos problemas (Fernandez, Insfran and Abrahão, 2011). Además, el conjunto de problemas de usabilidad que se pueden observar directamente en este tipo de análisis donde se comparan trazas de eventos es limitado. Por ejemplo, si una actividad es innecesariamente compleja y todos los usuarios la siguen de todas maneras, lo cual es posible cuando los usuarios son recurrentes y se adaptan a un mal diseño, no habría desvíos que observar, y el problema estaría aún presente.

Existen también muchas fuentes de buenas prácticas de usabilidad en la literatura, aunque es difícil para un desarrollador identificar cuáles de estas prácticas debe seguir ante un problema particular en una aplicación.

La propuesta presentada en este trabajo para resolver los problemas usuales del asesoramiento automatizado y solución de problemas de usabilidad, consiste en

enriquecer la captura y el análisis de eventos de interacción para poder reportar **problemas concretos**, que se puedan resolver mediante la técnica de **refactoring de usabilidad**. Al ser ésta una práctica ágil, permite mejorar la usabilidad incrementalmente, alimentándose de la interacción de los usuario, especialmente en aplicaciones que ya se encuentran corriendo en producción (Garrido, Rossi and Distante, 2011). Además, los refactorings son útiles, no sólo como soluciones mecánicas que pueden catalogarse, sino también debido a que cada solución está siempre metódicamente ligada al *bad smell* que resuelve. En el contexto de la usabilidad web, se empleará a lo largo del trabajo el término *usability smell*.

En la presente propuesta, se apunta a proveer asistencia automática tanto en la detección como en la aplicación de soluciones para problemas de usabilidad en aplicaciones web. La estrategia automatizada que se presenta está basada en hallar problemas caracterizados como *usability smells* mediante la captura y análisis de eventos de interacción de usuarios. Para conseguir esto, se trabajó sobre ideas previas en la literatura del campo, ligando eventos específicos a *usability smells*, definiendo así nuevos problemas y reportándolos en el momento en que aparecen, con un nivel de abstracción que hace posible sugerir soluciones concretas para ellos en términos de refactorings de usabilidad.

Para alcanzar el objetivo, se implementó una herramienta llamada **Kobold**. Esta herramienta funciona en realidad como un servicio, siguiendo la idea del esquema conocido como SaaS – *Software As A Service* (Software como servicio). En este caso, hablaremos específicamente de Usabilidad como Servicio. Kobold es una herramienta que funciona de manera mayormente desatendida, y sólo requiere un esfuerzo mínimo de instalación para ser incorporado en una aplicación web en producción. Esto hace que pueda ser utilizado por un amplio público de desarrolladores con diferentes grados de experiencia en usabilidad. En el caso particular de los expertos en el tema, la herramienta puede servirles para darles un rápido monitoreo de lo que sucede con la masa de usuarios reales. Por otro lado, aquellos que no tengan conocimientos específicos en usabilidad, podrán tener un diagnóstico concreto de los potenciales problemas e incluso sugerencias de solución.

1.1. Problemas de usabilidad web

Los problemas de usabilidad en los sitios o aplicaciones web no siempre son abordados, sobre todo en las organizaciones más pequeñas o no comerciales (por ejemplo, organismos del estado). Un posible motivo de esto, es la idea generalizada sobre el alto costo que tiene realizar un análisis de usabilidad, sumado a que en muchos casos se considera como un aspecto secundario. Sin embargo, la literatura sugiere lo contrario, tanto en lo respectivo a los costos como en la relevancia, al menos en el impacto comercial (Nielsen and Loranger, 2006; Gregg and Walczak, 2010).

Concretamente, los problemas de usabilidad son aquellos que impiden a los usuarios realizar tareas, ya sea en la búsqueda de contenidos ("*¿dónde está el teléfono de contacto?*") como en situaciones que requieren mayor interacción, es decir durante un *diálogo* con la aplicación, como por ejemplo al completar un pago o *checkout* en un sitio de e-commerce. Según la lista de Jakob Nielsen sobre los 10 errores de usabilidad web más frecuentes (*Top 10 Mistakes in Web Design*)[1], el problema principal son los formularios de búsqueda, seguido del mal uso de PDFs y otros problemas relacionados a presentación de contenidos, como tamaños pequeños de fuente o mala disposición del texto que dificulte la lectura.

Muchos de estos problemas pueden ser hallados mediante un análisis estático de las interfaces. En particular, problemas de contraste en los colores, fuentes utilizadas, estilo de los links, o tamaños en general de los elementos, pueden hallarse accediendo al código fuente y las hojas de estilo CSS. Otros problemas pueden encontrarse siguiendo directrices (*guidelines*) suficientemente concretas para ser automatizables (Dingli and Mifsud, 2011; Ivory, 2013; Torrente *et al.*, 2013). Sin embargo, muchos problemas sólo pueden detectarse observando a los usuarios interactuar son las aplicaciones. Por ejemplo, una directriz podría indicar que un formulario debe proveer mensajes de validación, pero esto sería imposible de evaluar automáticamente sin observar el uso del formulario. Los formularios son, de hecho, fundamentales en el análisis de usabilidad web, dado que son la herramienta principal de interacción con el usuario, donde se posibilita su participación en los procesos de la aplicación (Wroblewski, 2008).

[1] Top 10 mistakes in Web Design https://www.nngroup.com/articles/top-10-mistakes-web-design/

El rango de problemas que pueden hallarse observando la interacción de los usuarios, en contraste con el análisis heurístico, es muy amplio. Cualquier problema con el uso de los elementos de interfaz, como botones, links, campos de entrada o listas de selección, sólo puede observarse al revisar interacciones reales, ya sea mediante tests de usuario o inspección de *logs* de interacción.

1.2. Motivación

El asesoramiento de usabilidad automatizado para aplicaciones web podría ser de gran ayuda para los desarrolladores menos expertos, o con menos recursos. Más aún, si este asesoramiento automático logra incorporar información sobre la interacción de los usuarios, la cantidad y calidad de problemas a hallar sería mayor.

Una manera de obtener información sobre la interacción de los usuarios automáticamente es mediante la revisión de *logs* (bitácoras) de uso. Existen ya muchas metodologías que utilizan esta información (Atterer, Wnuk and Schmidt, 2006; Carta, Paternò and Santana, 2011) para reportar problemas, o al menos proveer visualizaciones y estadísticas. Idealmente, sería beneficioso poder contar con la máxima cantidad de información posible del uso real, para reportar problemas de usabilidad con el mayor nivel posible de abstracción, lo que permitiría que cualquier desarrollador pueda aprovecharlos y tomar medidas, sin importar su nivel de formación en usabilidad.

Aún si se pudiera encontrar la manera de buscar defectos de usabilidad web automáticamente con un buen nivel de abstracción, aún quedaría un segundo problema: resolverlos. En su libro “Don’t Make Me Think”, Steven Krug afirma que “siempre vas a encontrar más problemas de los que tus recursos te permitan resolver” (Krug, 2000). Teniendo en cuenta este punto de vista, contar con una herramienta de aplicación automatizada de soluciones a problemas de usabilidad podría ser de gran ayuda.

Si todo el proceso pudiera automatizarse, y además no requiriera de un esfuerzo de instalación considerable, se podría llegar a un amplio público de desarrolladores para mejorar sistemáticamente la usabilidad en sus aplicaciones web.

1.3. Automatización con usability smells y usability refactoring

En este trabajo se buscará llegar al máximo nivel de automatización en la búsqueda de problemas de usabilidad y sus soluciones. Para esto se utilizará la técnica de *refactoring*, una herramienta pensada originalmente para mejorar incrementalmente la calidad del código fuente (Fowler, 1999). Siguiendo esta idea, los problemas de usabilidad se definen como *usability smells* que pueden catalogarse y ligarse con *usability refactorings*, para facilitar su identificación y resolución.

Para automatizar la búsqueda de usability smells en tiempo real, se buscará analizar la interacción **real** de los usuarios finales, capturando y procesando eventos que contengan información específicamente de interés para hallar este tipo de problemas. Esto presenta múltiples desafíos: en primer lugar, es necesario definir heurísticas para detectar comportamientos que revelen mala usabilidad - y lograr que sean automatizables. Existen muchas directrices publicadas que identifican comportamientos específicos de los usuarios que muestran problemas de usabilidad, pero traducir estos comportamientos a reglas automatizables es todo un reto.

Las heurísticas de detección se basarán en el análisis de comportamiento de usuarios, con lo cual es necesario implementar una manera de capturar eventos que sea transparente para el usuario, y que facilite el análisis inmediato. Dado que la interacción se captura de usuarios reales, el número de eventos a analizar puede llegar a tomar grandes dimensiones, y es importante tener en cuenta la performance al momento del análisis.

No hay que olvidar que todo el análisis automático de eventos de interacción debe realizarse cuidando lo máximo posible la privacidad, dado que se observaría comportamiento de usuarios finales. Todos estos desafíos se discuten en el Capítulo 4 que describe la captura de eventos de usabilidad, y en el Capítulo 5 que detalla la captura de usability smells.

Para proveer soluciones automáticas se plantea el uso de refactorings de usabilidad, aprovechando el trabajo previo en Refactorings Web del Lado del Cliente (o CSWR por sus siglas en inglés – *Client Side Web Refactoring)* (Garrido, Firmenich, *et al.*, 2013). De esta manera, se puede seguir con la intención de ofrecer soporte a aquellos desarrolladores que no tengan necesariamente experiencia en usabilidad. Cabe destacar

que un experto podría aprovechar este aspecto de la herramienta de todas maneras, por simplicidad, o incluso para realizar pruebas. La búsqueda de automatización con mínimo esfuerzo de instalación presenta múltiples desafíos.

Un punto de partida para el desarrollo de refactorings automatizados son los catálogos de trabajos previos (Garrido, Rossi and Distante, 2011; Distante *et al.*, 2014). A la hora de programar estos refactorings, sin embargo, existe un problema en su nivel de especificidad: varios proponen soluciones en un nivel muy alto de abstracción, mientras que en esta propuesta se intenta ser lo más específico posible. Ligado directamente a este problema se encuentra el desafío de automatizar la generación y aplicación de los refactorings. Para la generación, la solución está en rediseñar los refactorings para hacerlos más detallados y concretos. Para la aplicación, el desafío es mayormente técnico. Gracias a la arquitectura propuesta, es posible aplicar refactorings sobre un sistema en producción gracias a la incorporación del script del lado del cliente, ya explicado al inicio de este capítulo.

Otro desafío a tener en cuenta está en la multiplicidad de soluciones que pueden resolver un mismo problema. Cuando un catálogo propone varias alternativas de refactoring para un mismo usability smell, generalmente hay uno que es más adecuado que los otros según el caso particular. Por ejemplo, consideremos que una interfaz incluye un campo de texto libre para ingresar una fecha, lo que se implicaría el smell *Unformatted Input*, ya que los usuarios tienen que encargarse de aplicar manualmente el formato a la fecha. Este problema puede resolverse al menos de tres maneras: incorporando un calendario (refactoring *Add Datepicker*) o agregando 3 listas de selección (refactoring *Turn Input into Selects*) para día, mes y año, o incorporando formato automático (*Format Input*). Dependiendo del tipo de fecha a ingresar, puede resultar mejor uno que otro. Una fecha de cumpleaños, por ejemplo, puede ser más fácil ingresar textualmente o mediante listas de selección o incluso con un campo de texto con asistencia de formato, ya que los usuarios generalmente la conocen de memoria. Para una fecha futura, como por ejemplo para una reserva de un hotel, podría ser más conveniente el calendario, ya que da un mejor contexto al usuario, y muestra los días de la semana. Todos los desafíos relacionados a los usability refactorings se detallan en el Capítulo 6.

1.4. Contribuciones

Concretamente, las contribuciones del presente trabajo son:

- Una estrategia para procesar eventos de interacción de usuarios “on the fly” y construir con ellos un modelo de *eventos de usabilidad* de más alto nivel, que capturan tanto la intención de los usuarios en su comportamiento como la semántica de los elementos de interfaz, y con ello permitan el posterior análisis de problemas concretos, así como información necesaria para la posterior reparación del problema.

- La definición de un catálogo de *usability smells de interacción de usuario*, algunos de ellos como refinamiento de usability smells más genéricos, y otros que surgen de observar la interacción real de usuarios en los sitios web, y que abstraen patrones de comportamiento y de estructura de páginas problemáticas, que pueden solucionarse a través de refactorings de usabilidad.

- Un catálogo de *usability refactorings* que pueden aplicarse del lado del cliente, donde cada refactoring se relaciona con el usability smell concreto que permite solucionar, identificando las transformaciones concretas que cada refactoring realiza.

- Un algoritmo para detectar la similitud entre elementos pequeños de una aplicación web, en la misma página o incluso entre páginas diferentes, y que permite agrupar los elementos que sufren el mismo usability smell y deben ser reparados por el mismo refactoring.

- Una herramienta que automatiza toda la propuesta e implementa el concepto de “Usabilidad como Servicio”, es decir, detecta usability smells en la interacción real de los usuarios de una aplicación web, sugiere posibles soluciones en términos de usability refactorings, y en muchos de los casos, aplica automática o semi-automáticamente dichas soluciones.

- Validaciones empíricas para todas las etapas del proceso, en las que se mide la factibilidad y utilidad tanto de la detección de usability smells, como de la aplicación automatizada de usability refactorings.

1.5. Organización de la Tesis

El resto del trabajo se organiza de la siguiente manera: el Capítulo 2 presenta el estado del arte en detección automática de problemas de usabilidad en aplicaciones web, así como nociones generales sobre usabilidad y refactoring.

El Capítulo 3 describe un resumen de la arquitectura presentada para la detección y corrección de problemas de usabilidad, explicando algunas decisiones de diseño y dando una introducción sobre cada uno de los pasos involucrados en el proceso. El Capítulo 4 detalla la detección de eventos de usabilidad, incluyendo un detalle uno por uno y explicando las dificultades de la implementación del lado del cliente. En el Capítulo 5 se profundiza sobre los usability smells, desde la definición hasta los desafíos técnicos de la detección, incluyendo un catálogo. El Capítulo 6 explica cómo se generan los usability refactorings, y se muestra un detalle de cada uno, junto con algunas notas sobre su implementación. También se provee una tabla completa que muestra la relación que existe entre todos los eventos de usabilidad, usability smells y usability refactorings.

En el Capítulo 7 se concentran todos los experimentos que se realizaron para validar cada uno de los pasos del proceso por separado, incluyendo un análisis preliminar sobre la utilidad de los usability refactorings en sitios de e-commerce. Finalmente, el Capítulo 8 detalla algunas conclusiones y propone trabajo futuro.

Capítulo 2. Trabajo relacionado

Este capítulo provee una revisión de diferentes metodologías relacionadas a la presente propuesta en distintos puntos. Primero se explican nociones sobre refactoring y *bad smells* en general, su aplicabilidad específica en el campo de usabilidad web, y el trabajo realizado sobre la definición de usability smells. Luego se exponen trabajos sobre evaluación de usabilidad en general y se coloca este trabajo en sus diferentes clasificaciones. Luego, se repasan trabajos relevantes de análisis de logs y visualización, siguiendo con tests de usuario remotos, para finalizar con los procesos que pueden compararse directamente con nuestra propuesta, y cuáles son las principales diferencias o ventajas respecto de los mismos.

2.1. Nociones de Usabilidad y Herramientas de Refactoring

El presente trabajo en usabilidad toma muchas ideas del área de **refactoring**. El concepto de refactoring fue originalmente definido por Opdyke como una transformación que preserva el comportamiento, apuntada a mejorar el diseño de código (Opdyke, 1992). Más adelante, Fowler popularizó la técnica al publicar un catálogo de refactorings para código orientado a objetos (Fowler, 1999). Los refactorings en el catálogo de Fowler buscan mejorar aspectos de la calidad interna del código, como comprensibilidad, extensibilidad y mantenibilidad de los diferentes componentes de un programa orientado a objetos, como clases, métodos y jerarquías, así como datos y expresiones condicionales. Un ejemplo de estos refactorings es “Extract Method” que convierte una pieza de código en un método independiente, con un nombre apropiado que explica su propósito.

El verdadero poder de la técnica de refactoring está en la ayuda que es capaz de proveer a los programadores menos experimentados para identificar potenciales problemas conocidos y aplicar una secuencia de pequeñas transformaciones para corregir esos problemas. En la jerga de refactoring, estos potenciales problemas se llaman “*bad smells*”, y su presencia significa que el código probablemente necesite refactoring. Por ejemplo, el refactoring “Extract Method”, presentado anteriormente, serviría para resolver smells como “Long Method” (Método Largo) y “Duplicate Code” (Código Duplicado) (Fowler, 1999).

La técnica de refactoring se convirtió en una práctica esencial para las metodologías ágiles, dado que los programadores pueden primero enfocarse en solucionar un problema con código, y luego mejorar su calidad sin alterar el comportamiento obtenido, en pequeños pasos seguros. El alcance de esta técnica se expandió rápidamente hacia otros paradigmas de programación y más allá de la mejora de cualidades internas de código, para mejorar cualidades externas (visibles) del software, como seguridad en bases de datos (Ambler and Sadalage, 2006), performance en programación paralela (Dig, 2011), y navegabilidad en aplicaciones web (Cabot and Gómez, 2008).

En 2007, se comenzó a trabajar en la aplicación de refactorings para mejorar la usabilidad de aplicaciones web (Garrido, Rossi and Distante, 2007). En ese trabajo se definió **usability refactoring** (refactoring de usabilidad) como un cambio en la

navegación, presentación o procesos de negocio de una aplicación web con el propósito de mejorar su usabilidad, a la vez que se preserva la funcionalidad esperada y los resultados (Garrido, Rossi and Distante, 2011). Con esos refactorings, los desarrolladores pueden obtener mejoras en la usabilidad, como por ejemplo una mejor distribución de contenidos en pantalla entre páginas, una mejor estructura de navegación y proceso, soporte adecuado para el usuario cuando está ejecutando procesos de negocio, etc. Un ejemplo de usability refactoring es "*Provide Breadcrumbs*", que ayuda a los usuarios a recordar el camino de navegación que siguieron hasta una página determinada, o la ubicación relativa de la misma (como se puede ver en el ejemplo de la Figura 1).

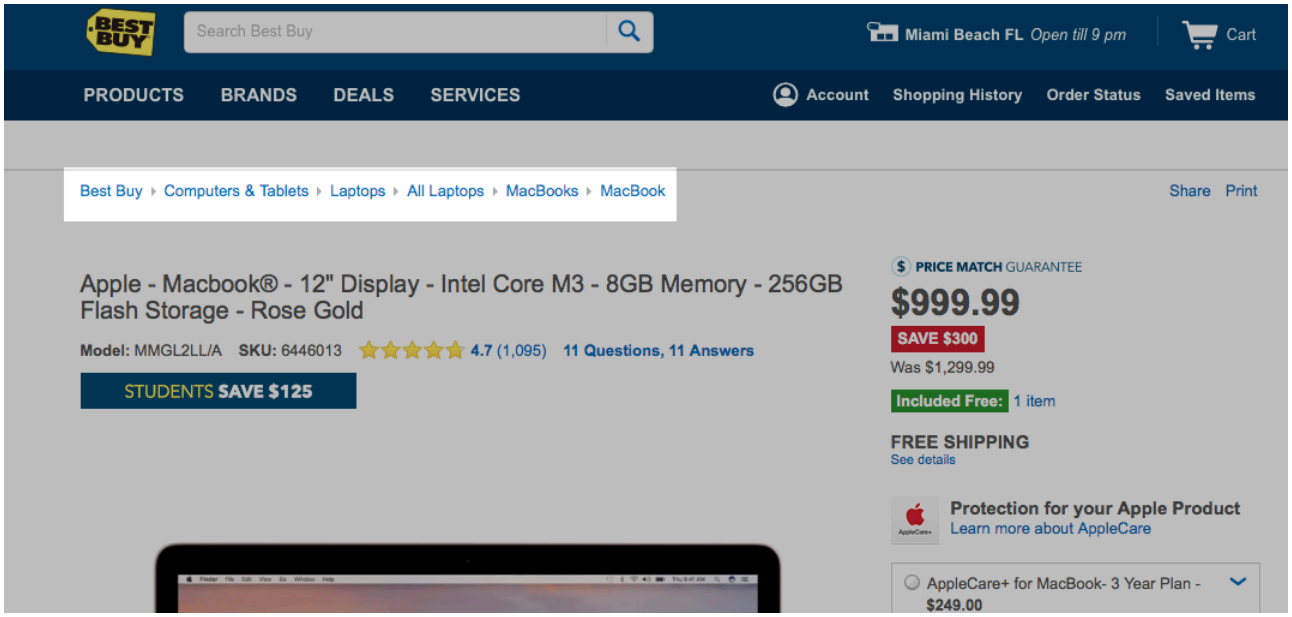

Figura 1: *Breadcrumbs* en bestbuy.com mostrando la ubicación relativa de la página de producto

De manera similar a los bad smells de código (o "*code smells*"), se definieron los **usability smells** (olores de usabilidad) como indicadores de posibles problemas de usabilidad que necesitan refactoring, donde estos problemas afectan en la usabilidad en uso de una aplicación web en alguno de sus aspectos: efectividad en uso, eficiencia en uso, o satisfacción en uso (ISO, 2011). En trabajos previos, los usability smells se catalogaron para ser reconocidos manualmente en los modelos de diseño de una aplicación web, a diferentes niveles (Garrido, Rossi and Distante, 2011). Ejemplos de estos smells son "Absence of meaningful links" (ausencia de links significativos) a nivel de modelo de navegación, "Cluttered interface" (interfaz cargada) a nivel de modelo de presentación, y "Long Activity" (actividad larga) a nivel del modelo de proceso.

En trabajos siguientes, se desarrolló un framework que permite aplicar refactorings de usabilidad del lado del cliente, reduciendo el costo de alterar un sistema en producción del lado del server (Garrido, Firmenich, *et al.*, 2013). También se desarrolló un catálogo con nuevos usability refactorings y usability smells (Distante *et al.*, 2014). Los refactorings en este catálogo pueden aplicarse tanto en el nivel de modelo de un proceso dirigido por modelos, o del lado del cliente, y los usability smells pueden ser descubiertos en los modelos o mediante inspección manual de los resultados de los tests de usuario. Un ejemplo de usability refactoring extraído de este catálogo es *Change the widget used to execute an activity* (Cambiar el *widget* empleado para ejecutar una actividad), apuntado a reemplazar un *widget* (elemento de interfaz) que se considera incómodo de utilizar o provoca errores en la interacción, por uno más apropiado. Un ejemplo de este refactoring sería reemplazar un campo de texto libre empleado para ingresar fechas por un selector de tipo calendario (como se muestra en la Figura 2), o cuando el campo de texto sirve para ingresar valores en un rango limitado, reemplazarlo por una lista tipo *select*. En el catálogo, el usability smell que dispara este refactoring es *Risk of Error*. Otros usability smells catalogados son *Difficult Access to information, User Confusion* y *Process inflexibility* (Distante *et al.*, 2014).

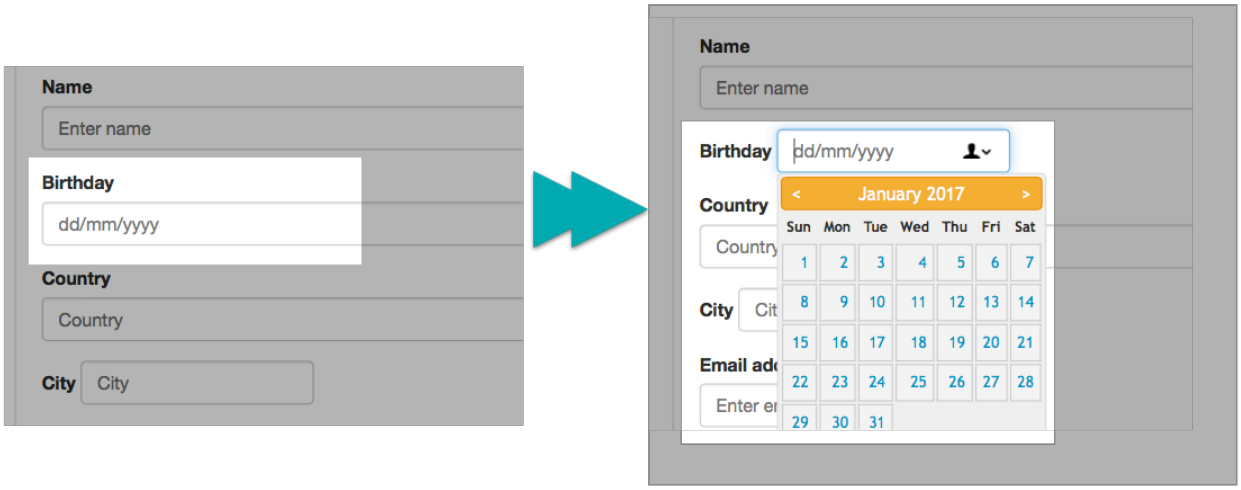

Figura 2: ejemplo gráfico de aplicación del refactoring *Change the widget used to execute an activity*, cambiando un campo de texto libre por un selector de fechas.

Además se han realizado tests estadísticos para medir la verdadera ventaja que ofrecen los refactorings en los aspectos de usabilidad en uso: efectividad en uso, eficiencia en uso y satisfacción en uso (Grigera *et al.*, 2016), con resultados positivos estadísticamente

significativos en la mitad de los refactorings evaluados para al menos un aspecto (en el Capítulo 7 se explicará en detalle). Para la preparación del experimento, la detección de usability smells no fue tarea sencilla, dado que requirió intervención manual de expertos para revisar los resultados de tests de usuario conducidos para tal fin.

Finalmente, en el contexto de refactoring, el proceso presentado en la presente propuesta para la detección de smells de usabilidad puede compararse con el trabajo llevado a cabo por Lanza y Marinescu para detectar sistemáticamente bad smells en código: ellos utilizan métricas conocidas del diseño orientado a objetos y patrones basados en métricas como estrategias de detección para identificar potenciales bad smells y problemas de diseño estructurales, y proveer los refactorings apropiados como solución (Lanza and Marinescu, 2006). De manera similar, en este trabajo se utilizan patrones de eventos de interacción para caracterizar usability smells y sugerir usability refactorings para la mayoría de los casos.

2.2. Tipos de métodos de evaluación de usabilidad

Como Hornbæk dijo en su revisión sistemática de 2006 (Hornbæk, 2006), la usabilidad no puede ser medida directamente, por lo cual los investigadores deben seleccionar aspectos de usabilidad que sí puedan ser medidos de alguna manera concreta, y representen indicadores válidos de usabilidad. En el método propuesto por Seffah et al. (Seffah *et al.*, 2006), por ejemplo, se crea un modelo de medición llamado “Quality in Use Integrated Measurement” (Medición Integrada de Calidad en Uso) en el cual se consideran 10 factores diferentes representando aspectos individuales de usabilidad definidos en algún estándar o modelo. Estos factores a se descomponen en 26 sub-factores que a su vez se descomponen en 127 métricas específicas. Esto denota la complejidad de tomar métricas directas de usabilidad, y la diversidad en los estándares publicados.

Existen diferentes maneras de clasificar métricas de usabilidad y métodos de evaluación de usabilidad. Una clasificación general bastante amplia para las evaluaciones es la propuesta por Fernandez et al. en 2011, y consiste en dos tipos fundamentales: **métodos de inspección** y **métodos empíricos** (Fernandez, Insfran and Abrahão, 2011). Mediante métodos de inspección, los evaluadores expertos realizan revisiones de concordancia y predicción de problemas basada en heurísticas. Por lo tanto, los métodos

de inspección están limitados por los tipos de problemas que pueden ser hallados en un ámbito de prueba de laboratorio, la calidad de las heurísticas y la experiencia del evaluador. Para conseguir resultados más rápidos, existen herramientas de análisis automatizado de heurísticas de usabilidad. Una de estas herramientas es USherlock (Cassino *et al.*, 2015). En una evaluación empírica, los autores demostraron la herramienta puede encontrar problemas con más efectividad (definida como la cantidad de problemas detectados sobre cantidad total de problemas presentes) que los métodos heurísticos manuales, representados por un conjunto de 24 evaluadores con experiencia media (estudiantes). Sin embargo, el tipo de problemas que pueden ser hallados con esta técnica están restringidos por dos limitantes fundamentales, por un lado, sólo pueden hallarse los problemas que no requieren razonamiento humano evidente (por ejemplo, claridad de contenidos), y por el otro, los problemas que pueden hallarse sin observar a los usuarios en acción, como niveles de contraste, tamaños de fuente, márgenes, etc. Las limitaciones de los métodos de inspección han conducido a la popularidad a los métodos empíricos, en particular los tests de usuario, que capturan y analizan datos de uso reales (Rubin and Chisnell, 2008). El proceso aquí presentado se basa en la idea de tests de usuario remotos, y los datos capturados en particular son eventos de interacción de usuario.

En el año 2000, Hilbert y Redmiles definieron un framework de comparación para caracterizar métodos de evaluación de usabilidad (Hilbert and Redmiles, 2000). Según este framework, el método aquí presentado se podría colocar en tres categorías: *Transformación*, dado que involucra seleccionar, abstraer y capturar cadenas de eventos, *Análisis*, en particular sumatorias y detección de secuencias, y *Visualización* de eventos abstractos (usability smells). En general, se provee un *Soporte integrado de evaluación*, es decir, un ambiente con soporte integrado para la composición flexible de varias transformaciones, análisis, y visualizaciones, si bien estas últimas no son el objetivo principal dado que se intenta dar un diagnóstico puntual con cada usability smell. También se analizan secuencias de acciones durante la detección de algunos usability smells, correspondiendo esto a la categoría de *Detección de secuencias*, pero no es parte central del método.

Otra revisión extensiva fue hecha por Ivory y Hearst en 2001 (Ivory and Hearst, 2001). Esta revisión presenta una taxonomía de métodos de evaluación de usabilidad

según su nivel de automatización, y clasifica un total de 132 métodos de evaluación en 4 dimensiones diferentes: Clase de Método, Tipo de Método, Tipo de Automatización y Nivel de Esfuerzo. En este trabajo, que tiene ya varios años, los autores advertían las múltiples ventajas de la automatización de métodos de evaluación de usabilidad, pero también señalaban que un reemplazo total de los métodos manuales no sería posible. Según esta taxonomía, el método presentado en este trabajo podría clasificarse como Clase de Método *testing*, dentro del Tipo de Método *log file analysis (análisis de archivos de log)*, con Tipo de Automatización *critique (crítica)*, y Nivel de Esfuerzo *minimal (mínimo)*.

Otra clasificación posible para los métodos de evaluación de usabilidad fue propuesta por Hartson y al. en 2003 (Hartson, Andre and Williges, 2003), que considera dos grandes tipos: formativos y sumativos. Los métodos de evaluación formativos son realizados por expertos en usabilidad al inicio de la etapa de diseño, para hallar problemas de usabilidad en prototipos anteriores a la puesta en producción. Por otro lado, los métodos sumativos se realizan luego del desarrollo, para asesorar o comparar el nivel de usabilidad alcanzados en el diseño final. Lo que se propone en este trabajo es un método automatizado sumativo, dado que se realiza sobre aplicaciones ya puestas en producción. La técnica de refactoring es particularmente relevante en este contexto, ya que propone realizar pequeñas mejoras incrementales para llegar a una mejora significativa sin comprometer otras partes del sistema.

Existen otros métodos que, al igual que la técnica de refactoring, proponen realizar mejoras incrementales. Genov definió la noción de *proceso iterativo de usabilidad* (*"iterative usability process"*) como una manera de obtener feedback continuo (Genov, 2005). Según el autor,

> *"…[el] rigor experimental no es necesario en todo test o investigación de usabilidad. Hay evaluaciones cuyo objetivo principal es ayudar a guiar el diseño y desarrollo de la interfaz, es decir testeo de usabilidad iterativo o de exploración"*

"…Experimental rigor is not necessary for all usability research/testing. There are evaluations whose main goal is to help guide interface design and development, namely iterative or exploratory usability testing"

Genov sostiene también que es relevante realizar tests con unos pocos usuarios para obtener feedback y hallar problemas de usabilidad sin la necesidad de grandes experimentos que involucren cientos o miles de usuarios en los que se busca aceptar o rechazar una hipótesis nula. Es importante destacar que el autor propone un método iterativo, aunque en etapas tempranas del desarrollo.

Entre los métodos destinados a incorporar usabilidad en etapas tempranas del desarrollo existen propuestas para integrar Aspectos Funcionales de Usabilidad (*Functional Usability Features* o *FUFs*) como primitivas en un método de Desarrollo Dirigido por Modelos (Model-Driven Development MDD) (Panach *et al.*, 2015). En esta misma línea, existen otros trabajos que incorporan análisis del comportamiento de los usuarios al proceso Dirigido por Modelos, como la propuesta de Bernaschina et al. (Bernaschina *et al.*, 2017), en la cual se presenta un framework de análisis empleando la técnica de *models at runtime*, que permite visualizar métricas relacionadas a la interacción directamente sobre los modelos de la aplicación. Existen también varios trabajos que intentan combinar Diseño Centrado en el Usuario (*User Centered Design* o *UCD*) con metodologías ágiles, ya que ambos procesos son iterativos y centrados en el usuario (Silva da Silva *et al.*, 2011; Jurca, Hellmann and Maurer, 2014). Los métodos ágiles proponen un proceso iterativo de desarrollo que involucra evaluaciones de diseño frecuentes y rediseño, centrado en el feedback de los usuarios (Williams and Cockburn, 2003), aunque no necesariamente se enfocan en la usabilidad del producto, que es el objetivo específico de UCD. Si bien la combinación de ambas metodologías no es trivial, estos trabajos demuestran que el objetivo de incorporar UCD en procesos ágiles es posible.

2.3. Análisis de logs y Métodos de Visualización

La idea de capturar métricas de usabilidad a partir de eventos de interacción ha sido una técnica empleada por mucho tiempo, incluso fuera del ámbito de la web. En 2002, cuando

la industria del análisis web estaba iniciando, investigadores como Chi ya desarrollaban sofisticadas técnicas de visualización y análisis de datos para ayudar a los expertos a comprender los grandes volúmenes de datos generados por los usuarios (Chi, 2002). Su método de visualización incluye una estructura basada en uso, patrones de tráfico y predicción de caminos comparados con caminos reales. De una manera similar a la empleada por las herramientas de analíticas web actuales, la visualización de Chi era capaz de señalar deficiencias en la navegación de un sitio, aunque requería de un experto en estadísticas de uso web para descubrir otros tipos de problemas concretos de usabilidad. Hoy en día, los servicios de analítica web provistos por herramientas comerciales como CrazyEgg[2], Hotjar[3], Mouseflow[4] o Woopra[5] que capturan automáticamente la interacción de los sitios en producción, ofreciendo mapas de calor y otras estadísticas que muestran visualmente el comportamiento de los usuarios (la Figura 3 muestra un mapa de calor –*heatmap*- típico de estas herramientas). Estos servicios demandan menos recursos, ya que los usuarios finales son quienes, sin saberlo, proveen los datos necesarios. Sin embargo, aún hay problemas concretos de alto nivel ocultos tras las estadísticas, y éstos requieren de un experto en usabilidad para descubrirlos y hallarles soluciones.

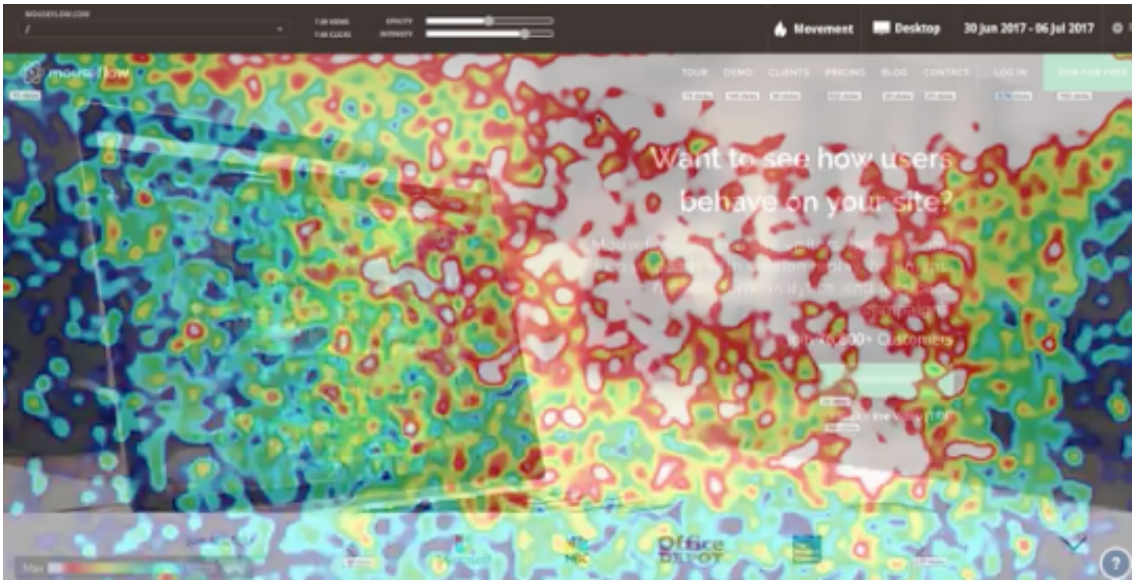

Figura 3: Mapa de calor (*heatmap*) de la herramienta *Mouseflow*, donde se muestra con colores la intensided de la actividad de mouse en las distintas áreas de la pantalla.

[2] Crazy Egg www.crazyegg.com
[3] Hotjar www.hotjar.com
[4] MouseFlow www.mouseflow.com
[5] Woopra www.woopra.com

En un trabajo de 2005, Saadawi et al. (Saadawi *et al.*, 2005) destacaron que los tests manuales tenían un alto costo en tiempo y recursos, y desarrollaron un sistema de captura de logs de interacción en sistemas desktop. Ese mismo trabajo contiene una validación interesante, que compara los resultados de tests realizados manualmente mediante la técnica de *think aloud* con los resultados capturados automáticamente por su herramienta para las mismas tareas. En su trabajo, los autores concluyen que, para un universo específico de problemas de usabilidad (fracaso al completar una tarea, tiempo excesivo en completar una tarea, múltiples intentos fallidos antes de terminar una tarea) la herramienta automática es capaz de capturar los mismos errores que un test *think aloud* (la herramienta capturó 10 de 12 problemas). La estructura del experimento en este trabajo es la misma que se realizó en la presente propuesta para comprobar la precisión de la captura automática de problemas de usabilidad.

Otro trabajo temprano de detección automatizada de problemas de usabilidad es el de Au et al. (Au *et al.*, 2008), que presenta un framework para detectar problemas de usabilidad relacionados al contexto específico de los dispositivos móviles. Si bien es un trabajo que tiene algunos años, y los dispositivos móviles han cambiado drásticamente desde entonces, varios de los problemas listados pueden aplicarse a los dispositivos de hoy. Por ejemplo, se describen problemas de precisión en links pequeños (que serían más graves hoy, dado que en 2008 se utilizaban los *stylus* como puntero para tocar la pantalla y hoy en día se usan los dedos).

En trabajos más recientes sobre el estudio de logs de interacción, la propuesta de Apaolaza et al. (Apaolaza, Harper and Jay, 2015) analiza registros de eventos de bajo nivel “en el llano”, es decir tomándolos de usuarios reales realizando tareas libremente en sitios reales. Ellos transforman estos logs en entidades de más alto nivel llamadas *micro-comportamientos*, y las utilizan para demostrar cómo el comportamiento de los usuarios evoluciona con el tiempo. En un trabajo más reciente, Apaolaza et al. presentan WevQuery (Apaolaza and Vigo, 2017), una herramienta que permite realizar análisis sofisticado de eventos de bajo nivel mediante la generación de consultas (queries) para hallar secuencias específicas. Las consultas se generan con un lenguaje visual y permiten realizar análisis en grandes volúmenes de logs de interacción previamente generados. A diferencia de lo que se propone en este trabajo (reportar problemas concretos para una audiencia amplia), queda en los expertos la tarea de exploración de los eventos mediante

consultas que, si bien constituyen una manera muy potente y flexible de observar la interacción, requiere de una mano experta. De manera similar, Breslav et al. (Breslav, Khan and Hornbæk, 2014) proponen descubrir *micro-interacciones* analizando sesiones de usuario grabadas en forma de eventos de interacción. En este trabajo también se capturan logs en producción, y se aprovechan las abstracciones de eventos de más alto nivel, pero no se almacenan sesiones completas. A diferencia de estas técnicas, en la presente propuesta se intenta trabajar sobre eventos en el momento en que se generan, y tales eventos ya se crean con un nivel más alto de abstracción desde el momento de su ocurrencia en el navegador del cliente, limitando en gran medida el tamaño final de la información almacenada y facilitando el análisis en muchos casos.

El trabajo de Shah et al. (Shah, 2008) propone también realizar análisis de eventos/logs de interacción para detectar patrones que indiquen que un usuario tiene dificultades al navegar. Por ejemplo, un indicador de dificultad sería la búsqueda constante de un contenido. Los eventos que se analizan son de bajo nivel, por ejemplo, movimientos, *clicks* y *scrolls* realizados con el mouse, o cargas de páginas (tiempos de carga, entrada, salida). En su trabajo recolectan, de diferentes autores, patrones de navegación o movimiento de mouse, y varios de ellos han sido utilizados en el presente trabajo. Algunos ejemplos de estos patrones son: *Cambio de Dirección*, que indica que un usuario abandona un curso de acción de una tarea; *Acción Cancelada*, que indica que un usuario vuelve rápidamente hacia atrás poco tiempo después de comenzar una acción (ej. Presionar el botón de “volver” del navegador, lo que sirvió de base de uno de los eventos de usabilidad diseñados para este trabajo); *Patrón de Dedos (Fingers Pattern)* que describe una sesión de navegación en la que siempre se vuelve a una página central, según algunos autores conocida como página “Hub” (Vigo and Harper, 2017). Shah et al. proponen incluso nuevos patrones y tratan de establecer una correlación entre los movimientos de mouse y su detección. Más allá de las diferencias en el nivel de los eventos, y el tipo de problemas que se busca, este trabajo está muy cerca metodológicamente de la presente propuesta, dado que no necesita especificar tareas a los usuarios para el análisis, y tampoco necesita patrones de entrenamiento para comparar la interacción recolectada.

En un trabajo similar, Okada y Fujioka proponen un método para detectar problemas de usabilidad a partir de trazas de mouse (operaciones innecesarias o fallidas),

mediante la comparación automatizada de logs de usuario contra logs óptimos de posiciones de click, capturados de tareas realizadas en forma óptima (Okada and Fujioka, 2008). Los autores reportan que con este método pueden encontrar 61% de los problemas encontrados manualmente en un tiempo mucho menor. La diferencia con la metodología presentada en este trabajo está en que no se requieren caminos óptimos para el análisis. Además, también se buscan otros tipos de problemas que los que se pueden hallar con la interacción basada en mouse.

Las metodologías listadas hasta este punto utilizan patrones fijos de comportamiento obtenidas de manera experimental, y luego intentan comprobar si efectivamente estos patrones se condicen con una mala experiencia de usuario. Otro enfoque interesante consiste en realizar los tests de usabilidad, observar las conductas problemáticas y *luego* extraer los patrones de comportamiento. Existen varios trabajos que realizan este tipo de experimento. Entre ellos, se puede mencionar la investigación realizada por Nakamichi et al. (Nakamichi *et al.*, 2006), donde se busca establecer una correlación entre malas experiencias de usuario (reportadas por ellos mismos) y su comportamiento capturado por múltiples vías, como movimientos de mouse, e incluso la posición de la vista capturada por un *eye tracker*.

Por otra parte, como resultado de un estudio realizado en 2016 sobre plataformas de educación a distancia, Harrati et al. (Harrati *et al.*, 2016) concluyen que la sola respuesta explícita de los usuarios en un análisis de usabilidad (mediante cuestionarios estandarizados, por ejemplo) no es suficiente, y que los signos que muestran durante su interacción ofrecen una visión mucho más amplia sobre la experiencia. La interacción evaluada por los autores consiste en métricas ligadas a la duración de las tareas, número de clicks y distancia del cursor. De estos trabajos se puede concluir entonces que (1) el análisis automatizado puede, en algunos casos, reemplazar con éxito el análisis manual, y (2) el análisis manual puede no ser tan exhaustivo como el automatizado, ya que no está limitado por tareas o por la subjetividad de los evaluadores. Si bien las pruebas manuales con expertos son al momento consideradas superiores a las automáticas en todo sentido exceptuando el costo, los investigadores citados coinciden en que hay un potencial para explotar.

Existen también trabajos que buscan problemas de usabilidad más específicos que vale la pena mencionar. Siendo la búsqueda un método tan importante a la hora de navegar (la lista de los 10 errores de usabilidad de Nielsen[6] ubica los problemas de búsqueda al tope de la misma) existen varios trabajos enfocados a la usabilidad en buscadores. Aula et al. (Aula, Khan and Guan, 2010) estudiaron el comportamientos de los usuarios en tareas de búsqueda para detectar cuándo tienen problemas para encontrar lo que buscan. En un extenso experimento con 179 participantes, los autores realizaron tests remotos con tareas para descubrir las conductas que indican dificultad en la concreción de dichas tareas, como por ejemplo tiempo de permanencia en páginas de resultado, o refinamiento de los términos de búsqueda. Los resultados son de interés para la propuesta aquí presentada ya que algunos de estos comportamientos pueden detectarse automáticamente.

2.4. Tests de usuario remotos

Los tests de usuario remotos ofrecen una gran ventaja sobre los presenciales, ya que permiten alcanzar una audiencia mucho mayor, y lograr ciertos niveles de automatización, con el costo de perder detalle en el análisis, y ciertos rasgos en la observación. Una de las primeras propuestas para realizar tests de usuario remotos fue hecha en 1998 por Hartson y Castillo (Castillo, Hartson and Hix, 1998). En su trabajo, desarrollaron un método que consistía en que los usuarios reales reportaran incidentes críticos. Desde entonces, muchas herramientas con esta misma metodología fueron creadas, pero agregando automatización en alguna de sus etapas.

Otros tipos de métodos de visualización para métricas de usabilidad han sido estudiados en diferentes metodologías que proponen tests de usabilidad remotos. WebQuilt (Hong *et al.*, 2001) es un sistema basado en *proxies* para correr tests de usabilidad controlados basados en tareas con usuarios voluntarios. Captura las visitas a páginas y caminos de navegación, así como también métricas basadas en tiempos, y muestra la información agregada de todas las sesiones en visualizaciones ampliables (*zommable visualizations*). Por lo tanto, si bien esta herramienta provee introspecciones valiosas en términos de tests de usuario, requiere un experto de usabilidad para crear las tareas, descubrir los problemas de usabilidad y arreglarlos.

[6] https://www.nngroup.com/articles/top-10-mistakes-web-design/

Atterer et al. proponen otra metodología para analizar logs durante sesiones de test de usuario remotos donde la información se captura mediante un proxy (Atterer, Wnuk and Schmidt, 2006). Mediante esta metodología capturan eventos de interacción de manera transparente y con suficiente detalle para que los resultados puedan ser utilizados en diferentes escenarios, desde tests de usabilidad, a caracterización (*profiling*) de usuarios y técnicas de visualización de datos de uso. Sin embargo, no detallan el proceso requerido para utilizar estos resultados en dichos escenarios. De una manera similar a ese trabajo, el método aquí presentado captura eventos de interacción de forma transparente al usuario, aunque lo hace con suficiente detalle como para ayudar a detectar usability smells, reduciendo el tiempo y la complejidad en el uso.

Herramientas más actuales en el área de tests de usuario remoto son WUP, WELFIT y CrowdStudy. De manera similar a WebQuilt, WUP (Web Usability Probe) (Burzacca and Paternò, 2013) utiliza un método basado en tareas para tests de usuario remotos, aunque en este caso se capturan eventos del lado del cliente. WUP utiliza sofisticadas visualizaciones para mostrar los eventos: un comparador de línea de tiempo que permita manipulación interactiva de líneas de tiempo y hallar desviaciones del camino óptimo, así como mostrar eventos clasificados por color y sus datos asociados; un comparador de flujo de navegación e incluso capturas de pantalla. En un trabajo más reciente, los autores presentaron una versión mejorada para comparación de líneas de tiempo (Paternò, Schiavone and Pitardi, 2016). La diferencia con el trabajo aquí presentado está en que trabajo con una lista específica de tares y la definición de un camino óptimo para cada una. Además, si bien provee técnicas automatizadas para descubrir desvíos del camino óptimo, sigue requiriendo de un experto en usabilidad para descubrir los problemas concretos en esas desviaciones, y sus soluciones requeridas.

En el caso de WELFIT (Santana and Baranauskas, 2015), los logs se visualizan mediante grafos de uso. Un grafo de uso puede verse como una combinación de caminos que representan la interacción de los usuarios. Cada nodo de estos grafos representa un evento activado en un elemento de una página web. Al igual que WUP, la herramienta WELFIT requiere de un experto para encontrar desviaciones en los caminos presentes en el grafo, y así descubrir problemas de usabilidad en un grafo potencialmente muy extenso.

Otro trabajo para organizar tests remotos de usuario y capturar sus resultados es CrowdStudy (Nebeling, Speicher and Norrie, 2013a). CrowdStudy presenta un abordaje interesante a los tests remotos de usuario, ya que incorpora herramientas de *crowdsourcing* como el *Mechanical Turk* de Amazon[7], una importante plataforma para reclutar voluntarios, y también considera su contexto, como el tipo de dispositivo empleado. También presentan un conjunto de métricas para obtener los resultados de los tests. Aunque es una metodología bastante diferente de la presentada en este trabajo, al estar basada en tests de usuario con tareas guiadas, la arquitectura de CrowdStudy es interesante para lo que aquí se desarrolla, dado que también se intenta obtener métricas de usabilidad a partir de una masa de usuarios lo más grande posible. En un trabajo previo por parte de algunos de los autores (Speicher, Both and Gaedke, no date), se presenta un framework para predecir métricas de usabilidad basadas en interacción de los usuarios. Sin embargo, requiere de datos de entrenamiento y de feedback explícito por parte de los usuarios mediante cuestionarios, todo antes de comenzar a colectar eventos de interacción reales.

2.5. Evaluación automatizada de logs en ámbitos no controlados

La búsqueda de problemas de usabilidad web en ambientes no controlados requiere diferentes técnicas de análisis debido a que no hay cadenas de eventos óptimas contra las cuales comparar, y la cantidad de datos es potencialmente muy grande y desorganizada. Además, al no haber tareas especificadas, se torna más difícil conocer el propósito de los usuarios. Sin embargo, existen varios beneficios sobre los métodos controlados: en primer lugar, son menos costosos dado que no requieren reclutar voluntarios ni diseñar tareas. Por otra parte, pueden hallar problemas que sólo aparecen en un ambiente real, donde los usuarios no están restringidos a realizar tareas fijas que sólo cubren parte de la funcionalidad disponible (a menos que sean diseñadas cuidadosamente para cubrir todo, pero rara vez es el caso). Por último, es importante destacar que los usuarios no están condicionados, sino que están realizando sus tareas libremente, y por lo tanto se captura más fielmente lo que sucede en realidad durante las visitas.

[7] https://www.mturk.com/mturk/welcome

El trabajo de Speicher et al. (Speicher, Both and Gaedke, 2015) presenta un conjunto de herramientas para obtener puntajes de usabilidad en páginas de resultados de búsqueda (SERPs – *Search Engine Results Pages*) y reparar los problemas hallados. A pesar de estar enfocado sólo en SERPs, el método también registra eventos de interacción para medir usabilidad y provee sugerencias de mejoras obtenidas de un catálogo de buenas prácticas. Una vez que una nueva versión de un SERP se crea manualmente siguiendo las sugerencias de la herramienta, se realiza un análisis comparativo entre la vieja y la nueva versión mediante un ciclo de A/B testing. Los puntajes relativos a la usabilidad se calculan según 7 factores diferentes, como comprensibilidad, distracción o densidad de información.

Otro trabajo similar al que se propone aquí es el de Harms et al. (Harms and Grabowski, 2014), que también emplea el concepto de usability smell para señalar problemas de alto nivel en elementos específicos de interfaces web. Utilizan una definición de usability smell levemente diferente de la que se presenta en este trabajo, refiriéndose a ellos como *comportamientos de usuario excepcionales que pueden indicar un problema de usabilidad*. Existen dos diferencias fundamentales con este enfoque: primero, la forma de los logs capturados. Ellos obtienen largas tareas de sesiones completas de usuarios, y generan árboles de tareas para detectar los smells, mientras que en este trabajo se intenta proveer resultados inmediatos, apenas aparecen los problemas. La segunda diferencia se deriva de la primera, y consiste en la naturaleza de los smells, que están más orientados a descubrir ineficiencias en el flujo de las tareas. A pesar de estas diferencias, ambos enfoques descriptos anteriormente pueden proveer buenos complementos para las ideas aquí presentadas, ya que los procesos de detección son bastante diferentes, pero también automatizados, y con eso sería posible ampliar el espectro de problemas a detectar o corregir.

En un amplio estudio realizado en 2017, Vigo y Harper (Vigo and Harper, 2017) evalúan diferentes patrones de navegación que representan potenciales indicadores de confusión en los usuarios, basados en literatura previa y en experimentación propia. El estudio, si bien está realizado con voluntarios, busca validar estos patrones para detectar comportamiento errático sin especificar tareas. Esto permitiría que, según los resultados, tal proceso de detección pueda ser implementado luego de la validación para analizar el comportamiento de los usuarios reales, eliminando por un lado la necesidad de reclutar

voluntarios, y obteniendo resultados inmediatos por otro (de la misma forma de lo que se busca en el presente trabajo). Los patrones empleados buscan conductas “adaptativas”, donde un usuario está esforzándose notablemente por llevar a cabo alguna tarea, ya sea porque está perdido en un camino de navegación, volviendo siempre a una misma página (empleada como lugar “seguro”), o que parece estar buscando ayuda. Otro resultado particularmente interesante de este trabajo es la posibilidad de clasificar automáticamente los estilos de navegación según dos criterios: el tipo de sitio (por ejemplo, si se trata de una aplicación web, o una aplicación de consumo de contenido), y el propósito particular del usuario (por ejemplo, si está determinado a realizar una tarea, o sólo mirando contenido sin un objetivo). El trabajo en sí provee una valiosa introspectiva sobre el análisis automatizado de patrones de navegación. Una limitante para su aplicabilidad está en que la detección se realiza con un complemento de navegador (Firefox en particular), aunque podrían hallarse maneras alternativas sin restringir la instalación a la intervención del navegador. Además, no propone soluciones para los problemas hallados.

El enfoque más cercano al que aquí se propone en términos de funcionalidad es, según la investigación realizada en la bibliografía, W3Touch (Nebeling, Speicher and Norrie, 2013b). Se trata de un conjunto de herramientas para evaluar problemas de interacción en interfaces táctiles de dispositivos móviles, y proveer adaptaciones para resolverlos. Incluye complejos mecanismos para registrar eventos de interacción táctil y métricas específicas para estos eventos, que por defecto son links errados (*missed links*) y nivel de zoom (*zoom levels*). W3Touch provee adaptaciones por defecto para algunos de los casos que consisten en alterar el tamaño y espaciado de los elementos afectados, dependiendo del dispositivo, y permite disparar adaptaciones creadas por desarrolladores manualmente. La herramienta analiza datos registrados e identifica componentes críticos, como ser elementos DOM que probablemente tengan problemas relacionados a la interacción táctil, es decir *missed links* o *zoom levels* más allá de umbrales establecidos. Este análisis puede ser comparado con el proceso que en la presente propuesta se aplica al clasificar elementos con potenciales problemas de usabilidad. Sin embargo, para agrupar elementos similares, W3Touch utiliza una clasificación basada en identificadores de elementos DOM. De esta manera logran segmentar la página en componentes críticos. La presente propuesta, en cambio, utiliza un algoritmo de similitud para clasificar elementos DOM en términos de equivalencia semántica y, por lo tanto, es capaz de

determinar situaciones en las cuales elementos levemente diferentes, pero equivalentes para el usuario (como por ejemplo diferentes productos de un listado), sufren del mismo smell, y por lo tanto pueden ser reparados aplicando la misma solución. Fuera de esta distinción más bien técnica, la principal diferencia con lo que se propone en este trabajo radica en que W3Touch está esencialmente enfocado en *responsiveness*, es decir la capacidad de las interfaces de adaptarse a diferentes dispositivos táctiles, en vez de usabilidad web en general. Además, W3Touch es capaz de detectar problemas de interacción en páginas aisladas, mientras que el enfoque aquí propuesto es capaz de capturar smells de alto nivel que pueden involucrar navegaciones a través de diferentes páginas, ampliando el espectro de problemas detectables.

Capítulo 3. Arquitectura

La mayoría de las herramientas para buscar problemas de usabilidad automáticamente que hay en la literatura requieren de un proceso de instalación. Además, un grupo considerable de ellas sólo están pensadas para ser utilizadas en tests con voluntarios. Si bien la mayoría son herramientas experimentales, el enfoque de las mismas relega la simplicidad en la adopción.

En la propuesta presentada en este trabajo, la facilidad de adopción fue considerada como un factor de importancia desde el inicio, y por este motivo se optó por proveer la solución para la detección y corrección de problemas de usabilidad como un **servicio**. En este modelo, los usuarios desarrolladores registran una cuenta, y luego de incorporar pequeño código en la aplicación web a analizar, el sistema comienza a capturar la interacción de los usuarios finales para hallar problemas de usabilidad. Cuando hay suficientes eventos, los problemas de usabilidad que se hallan son

reportados en forma usability smells junto con las sugerencias para resolverlos en forma de refactorings de usabilidad.

Este mecanismo se define en este trabajo con el término “Usabilidad como Servicio”, como un caso específico de “Software como Servicio” o SaaS (por el acrónimo del inglés “Software as a Service”). Un esquema de la arquitectura general del servicio puede observarse en la Figura 4.

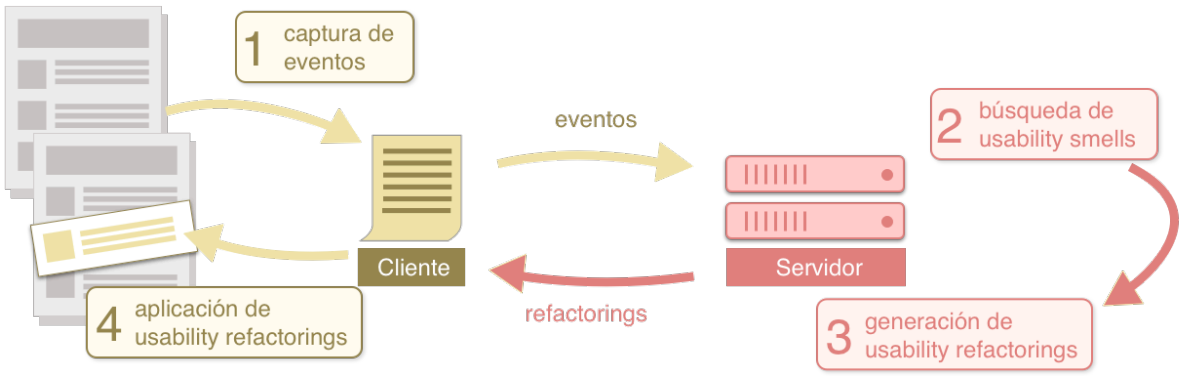

Figura 4: Arquitectura general de la propuesta

Parte de la implementación depende de un componente que funciona del lado del cliente, el cual los desarrolladores incluyen en su aplicación al momento de registrarse, siendo el único requisito para que el servicio funcione. Este componente tiene un doble propósito: por un lado, captura los eventos del usuario para interpretar su interacción (paso 1 de la figura), y por otro lado hace posible intervenir la interfaz para aplicar refactorings de usabilidad cuando el desarrollador así lo decide (paso 4 de la figura).

El resto del procesamiento sucede en el servidor. Los eventos que captura el componente en el cliente son procesados allí para detectar los usability smells (paso 2). Cada usability smell se reporta inmediatamente junto con sugerencias de refactorings, ya sean éstos aplicables automáticamente o sólo descriptos para ser implementados de forma manual por los desarrolladores. Cuando un usuario desarrollador decide aplicar un refactoring de usabilidad automatizado, éste se instancia (también en el servidor – paso 3), generando el código JavaScript a ejecutar para modificar la interfaz de la aplicación web. Este código es invocado por el componente que corre en el cliente cada vez que aparece el componente de interfaz afectado por el usability smell.

A continuación, se describen dos decisiones importantes en el diseño de la arquitectura: por un lado, la estructura de Usabilidad como Servicio, y por el otro, el componente del lado de cliente que interviene tanto en la captura de eventos como en la aplicación de refactorings.

Usabilidad como un Servicio

La decisión principal de diseño para la propuesta consiste en proveer el análisis de usabilidad en forma de servicio. Esto tiene varias ventajas y algunos desafíos; todos ellos fueron analizados para tomar la decisión.

Como se explicó anteriormente, la ventaja más determinante para optar por este tipo de implementación fue la simpleza para los usuarios, ya que el foco del proceso siempre estuvo puesto en facilitar al máximo la adopción. Dado que la propuesta es bajar al mínimo posible los costos del análisis y reparación de problemas de usabilidad, no tiene sentido imponer un esfuerzo significativo de instalación a los usuarios del sistema. Por este motivo, se buscó la manera más simple posible, brindando un servicio web en el cual los usuarios registran una cuenta y luego ingresan para ver los resultados. Existe un solo paso adicional importante: incorporar un *snippet* de código en la aplicación analizada para permitir que el servicio recolecte la interacción de los usuarios finales. Todo el proceso es muy similar al de otros servicios como Google Analytics, por citar un ejemplo.

Pre-Procesamiento del lado del cliente

Una parte fundamental del proceso propuesto es llevada a cabo por un componente que se ejecuta en el navegador del usuario final. Este componente programado en JavaScript cumple dos funciones importantes: captura los eventos del usuario, y aplica los refactorings de usabilidad.

La captura de eventos juega un papel fundamental en el proceso por la manera en que se realiza. Los eventos de bajo nivel son evaluados y compuestos para generar eventos de mayor nivel de abstracción llamados *usability events* o eventos de usabilidad. Esto alivia en gran medida la carga del servidor y permite una mayor riqueza en el nivel de información sobre la interacción de los usuarios.

La aplicación de refactorings también es llevada a cabo por el mismo componente. Gracias a que se encuentra embebido en la aplicación bajo análisis, le es posible modificar su interfaz. Para esto, se comunica primero con el servidor con el fin de obtener los refactorings que se deben aplicar, y luego los instala.

Pasos del Proceso

La estrategia empleada consiste en tres pasos: Captura de Eventos, Detección de Usability Smells y Refactoring. Todo esto sucede en un esquema combinado de componentes basados en el cliente, es decir en el navegador web, y componentes en un servidor, en referencia a una aplicación remota, diferente de la que se estudia. La captura, por ejemplo, sucede mayormente del lado del cliente, mientras que el análisis para la detección de usability smells se realiza en un servidor. En la etapa de captura de eventos, el componente del lado del cliente se ocupa de analizar los eventos que se generan durante la interacción de los usuarios, como clicks o teclas presionadas, para filtrarlos, agruparlos y convertirlos en eventos más abstractos y con más información llamados **eventos de usabilidad**. Durante la etapa de detección de usability smells, el componente del lado del servidor clasifica y analiza eventos de usabilidad utilizando algoritmos especializados para descubrir problemas de usabilidad de diferentes tipos. Finalmente, en la etapa de refactoring, puede haber diferentes resultados según el *smell*. Cada smell puede tener uno o más refactorings que lo resuelven. Dado un refactoring, si no es posible automatizar su aplicación, entonces se ofrece como una sugerencia. Si la automatización es posible, y el usuario acepta aplicarla, se genera el código necesario en el componente del servidor y se inserta en la aplicación web utilizando el componente del cliente.

Es importante destacar que todos los componentes de la arquitectura forman parte de un framework que permite su fácil extensión. En el caso de la detección de eventos de usabilidad del lado del cliente, la incorporación de nuevos eventos está facilitada mediante múltiples funcionalidades que simplifican la captura y manipulación de xpaths y la conexión con el servidor. Al estar basado en el framework de JavaScript jQuery[8], el manejo de eventos y elementos DOM también está simplificado. Las otras dos partes del

[8] jQuery https://jquery.com

proceso están implementadas del lado del servidor, en Pharo Smalltalk[9]. En la etapa de detección de usability smells, la definición de nuevos buscadores (es decir, de nuevas heurísticas para cada smell) es más sencilla, ya que cuenta con un mecanismo reutilizable para procesar eventos. La etapa de usability refactorings cuenta con lógica implementada para la generación de código JavaScript que también simplifica la tarea. El código del proyecto está abierto a la comunidad en un repositorio GitHub[10]. Una versión completa de la herramienta está disponible online[11].

3.1. Captura y análisis de Eventos de Usabilidad

La captura de eventos como estrategia para encontrar problemas de interacción ha sido ampliamente utilizada en el campo de la usabilidad (Ivory and Hearst, 2001). Muchas técnicas simplemente capturan los eventos del lado del cliente y luego los procesan en el servidor, para proveer sofisticadas visualizaciones, o en algunos casos métricas que computan desvíos a partir de una interacción tomada como referencia, que marca la manera óptima o más corriente de realizar una tarea.

Durante la investigación realizada en usabilidad web en este trabajo, se observó que detectar usability smells manualmente sobre aplicaciones web era una tarea costosa, pero que podría ser automatizada en cierto nivel empleando técnicas conocidas de análisis sobre eventos de interacción. Es importante destacar, sin embargo, que en este trabajo se intenta proveer a los desarrolladores con reportes en tiempo real, es decir apenas se manifiesta un problema, y procesar enormes cantidades de logs usando heurísticas o algoritmos en el servidor puede ser una tarea que lleve demasiado tiempo. Por este motivo es que se realiza un análisis preliminar del lado del cliente, filtrando y agregando eventos de interacción de bajo nivel (JavaScript) y estos nuevos eventos pre-procesados sólo se envían al servidor si son potencialmente útiles para detectar usability smells. Además, realizando procesamiento del lado del cliente, se puede recolectar información relativa al contexto en el que se encuentra el usuario en el mismo momento en que el evento está sucediendo. En consecuencia, la agregación de eventos de

[9] Pharo Smalltalk http://pharo.org
[10] Kobold – Proyecto en Github https://github.com/juliangrigera/Kobold
[11] Kobold http://autorefactoring.lifia.info.unlp.edu.ar

interacción de bajo nivel, junto con la información contextual sobre el evento, permite generar eventos de más alto nivel, los cuales llamamos **eventos de usabilidad**.

Los eventos de usabilidad fueron diseñados a partir de los usability smells. Esto significa que primero se observaron y estudiaron los comportamientos que indican la presencia de un usability smell, y luego se descompusieron estos comportamientos en acciones más atómicas que pudieran ser capturadas de un usuario en una sesión normal de trabajo. Un ejemplo de evento de usabilidad es “Flash Scroll”. Cuando un usuario desplaza verticalmente un contenido web (ya sea con la barra de scroll o el mouse) más rápidamente que una velocidad determinada, se captura un evento Flash Scroll, que agrega toda la información relevante al mismo, como posición vertical al comienzo y al final, velocidad en pixels / segundo y *timestamp* (fecha y momento preciso del evento).

Todos los eventos de usabilidad generados por los usuarios son recolectados y analizados en el servidor, en el segundo paso del proceso. En este caso, si suficientes usuarios disparan el evento Flash Scroll sobre el mismo conjunto de elementos de interfaz, esto indicará la presencia del usability smell llamado “Overlooked Contents” (contenidos ignorados). Esto puede ser un signo de que cierto contenido en una página no es interesante para la mayoría de los usuarios.

Existe una relación muchos-a-muchas entre los eventos de usabilidad y los usability smells: un evento de usabilidad puede servir para detectar más de un usability smell, y en el caso opuesto, detectar un usability smell puede requerir del análisis de más de un tipo de evento de usabilidad.

3.2. Captura de Usability Smells

Este paso consiste en clasificar, agregar y analizar los eventos de usabilidad capturados para descubrir usability smells. Dado que el análisis para detectar diferentes smells varía de uno al otro, hay un algoritmo diferente para cada uno. Estos algoritmos están implementados en entidades llamadas *usability smell finders* (buscadores de usability smells), y cada uno de estos buscadores es capaz de detectar un único tipo de usability smell, consumiendo y analizando uno o más tipos de eventos de usabilidad. Cada aplicación bajo análisis tiene su propio conjunto de buscadores, y cada buscador está configurado con ciertos parámetros que determinan el número, proporción, o combinación

de eventos de usabilidad que desencadenan la presencia de un smell específico. Los valores por defecto fueron hallados mediante experimentación con la herramienta, pero se pueden ajustar para servir propósitos específicos.

Para producir resultados en tiempo real, los eventos de usabilidad se procesan en el momento en el que llegan al servidor, y todos los buscadores de smells individuales re-evalúan los eventos que acumularon para detectar potenciales nuevos usability smells. Para acelerar este proceso de detección, se ajustó su ejecución para que no dependiera de la cantidad de eventos acumulados. Esto sucede en 3 etapas:

1. **Clasificación de Eventos:** al momento de arribar al servidor, el nuevo evento de usabilidad se envía a todos los buscadores asociados. Por ejemplo, si el evento que llega es un *Click Attempt*, se envía al buscador que detecta el smell *Unresponsive Element*. Una vez que el evento llega al buscador, se reclasifica una vez más, esta vez según el elemento afectado (generalmente un elemento DOM). Esto permite reevaluar sólo los eventos que afectan a este elemento.
2. **Síntesis de Datos:** no bien el evento se clasifica por el elemento afectado, el buscador extrae cierta información. Por ejemplo, puede incrementar un contador o actualizar un promedio almacenado en *cache*. Esta síntesis es clave para la performance del proceso, dado que evita volver a recorrer todos los eventos cada vez que uno nuevo arriba.
3. **Evaluación de Usability Smells:** con la información actualizada por el nuevo evento, el buscador reevalúa la presencia de un smell sobre el elemento afectado. Nuevos smells se agregarán al reporte, y los smells previos que se determine que ya no están presentes (debido a que los nuevos valores ya no lo indican) se guardan para mantener una consistencia, pero se indica que ya no están presentes en el reporte.

Esta manera de procesar la información, en especial la síntesis de datos, permite obtener resultados instantáneos sin depender de la cantidad de usability events acumulados. Estos tres pasos que componen la detección de usability smells se ilustran en la Figura 5. En la misma, los círculos representan usability events que viajan desde el cliente al componente del servidor, donde son clasificados y almacenados para detectar usability smells.

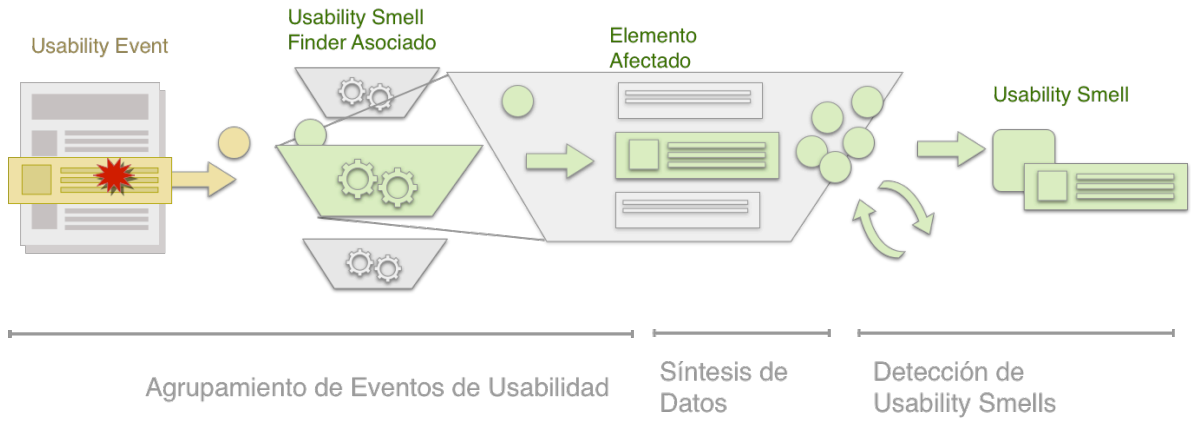

Figura 5: Proceso de detección de usability smells

El conjunto de buscadores actuales puede extenderse fácilmente. La arquitectura del sistema está preparada de tal manera que agregar un buscador para una nueva clase de usability smell sólo implica seleccionar el tipo o los tipos de usability event que deben ser consumidos, el criterio para agruparlos (se pueden elegir varias estrategias implementadas o implementar una nueva, por defecto es el mismo elemento DOM), y la lógica de detección, que típicamente involucra contar proporciones sobre ciertas propiedades de los eventos.

A continuación, se ilustrará el proceso de detección con dos usability smells: *Scarce Search Results* y *Free Input for Limited Values*, mostrando una breve descripción para cada uno, los eventos que los producen, y cómo estos eventos son procesados. Adicionalmente se enumerarán los parámetros que pueden ser ajustados para el proceso de detección y los refactorings sugeridos para resolver el usability smell.

Scarce Search Results

- **Descripción:** un formulario de búsqueda usualmente no trae resultados.
- Usability Events: Search
- Proceso de detección:

1. Para cada formulario afectado por los usability events, se dividen los eventos en dos grupos: aquellas búsquedas con resultados, y las que no trajeron resultados.
2. Si el grupo de búsquedas sin resultados es proporcionalmente mayor que el otro grupo por un margen superior a un umbral, el usability smell se reporta para el formulario en cuestión.

- Parámetros ajustables:

- Proporción de mayoría de búsquedas fallidas: ¿cuál es la poporción mínima de búsquedas sin resultados que indican la presencia del smell?

- **Refactorings recomendados:** *Add Autocomplete* para el formulario de búsqueda, disminuyendo la incertidumbre de los usuarios antes de pulsar el botón de “Buscar”

En este caso la lógica es simple, pero el script del lado del cliente juega un papel mayor en la recolección de los usability events *Search*, dado que allí es donde se determina si una búsqueda tuvo éxito o no. Y esto debe estar preparado para funcionar en una gran variedad de interfaces.

Free Input for Limited Values

- **Descripción:** un campo de texto libre se presenta al usuario, pero los valores aceptados pertenecen a un conjunto limitado.
- Usability Events: Text Input
- Proceso de detección:

1. Para todos los eventos sobre un campo de texto determinado, se agrupan los que tienen el mismo valor, o similar. Para determinar la similitud, se utiliza el algoritmo de distancia de edición de Levenshtein (Levenshtein, 1966). Agrupar los valores restantes en un grupo llamado “otros”.
2. Contar los eventos para cada grupo.

3. Si (a) el tamaño de cada grupo excepto “otros” es mayor que un umbral mínimo de proporción, y (b) la proporción del grupo “otros” es más pequeña que un tope de tolerancia, se reporta el smell.

- Parámetros ajustables:

  - Proporción de mayoría de búsquedas fallidas: ¿cuál es la poporción mínima de búsquedas sin resultados que indican la presencia del smell?

- Refactorings recomendados:

  - *Add Autocomplete* para el formulario de búsqueda, disminuyendo la incertidumbre de los usuarios antes de pulsar el botón de “Buscar”.

  - *Text Input into Radios* reemplaza la entrada de texto por un grupo de *radio buttons*, donde cada opción es más visible. Además, se agrega una opción “otro” para que se pueda ingresar un texto diferente de las opciones predefinidas. Este refactoring sólo se recomienda cuando los valores frecuentes son pocos (hasta 8 por defecto).

La lógica de detección para este smell recae más en el análisis de los eventos, algo más complejo que en el ejemplo anterior.

3.3. Sugerencia y Aplicación de Refactorings

A partir de los problemas hallados en la etapa de detección de usability smells, es posible sugerir soluciones en forma de refactorings de usabilidad, para que los desarrolladores puedan ir aplicando soluciones incrementalmente a los problemas reportados. Gracias a los catálogos preexistentes que sirvieron como punto de partida (Garrido, Firmenich, *et al.*, 2013; Grigera *et al.*, 2016), conectar los problemas (usability smells) con las soluciones (usability refactorings) no es una tarea compleja.

Las soluciones en términos de refactoring pueden generarse aprovechando el trabajo previo sobre Refactorings Web del lado del Cliente (o CSWR por sus siglas en inglés: *Client Side Web Refactorings)* (Garrido, Firmenich, *et al.*, 2013). De este modo, es posible modificar una aplicación web sin necesidad de alterar o siquiera conocer el código

existente del lado del servidor, ya que todo el trabajo de transformar la interfaz se realiza utilizando código HTML o JavaScript.

La información requerida para generar automáticamente el código de los usability refactorings surge de los mismos usability smells que intentan solucionar. Por ejemplo, el smell *Free Input for Limited Values* explicado en la sección anterior puede solucionarse aplicando un *Add Autocomplete*, que es un refactoring que agrega a un campo de texto libre la capacidad de sugerir términos para autocompletado. En este caso, los valores que se sugieren en el autocompletado se obtienen del listado de valores frecuentes que ingresaron los usuarios antes de detectar el usability smell.

Capítulo 4. Eventos de Usabilidad

La parte central de la metodología está en la captura e interpretación de la interacción de los usuarios. Esta tarea es particularmente desafiante, teniendo en cuenta que la interacción se captura en aplicaciones en producción, con usuarios reales, y sin tareas predefinidas.

Este capítulo describe como se pre-procesan los eventos de interacción, eliminando eventos o atributos irrelevantes, filtrando y agrupando sólo aquellos eventos que son de utilidad para el propósito de reconocer patrones de interacción defectuosos o sospechosos de producir problemas de usabilidad.

La segunda sección del capítulo describe el catálogo de eventos de usabilidad que se definen en esta propuesta.

4.1. Pre-Procesamiento de Eventos

Como se mencionó en el Capítulo 3, los eventos de interacción se analizan y capturan en el cliente, a medida que ocurren. La captura de eventos se realiza mediante JavaScript. Los eventos de bajo nivel, como movimientos de mouse o presionado de teclas, permiten detectar un amplio rango de acciones, pero guardar todos y cada uno de ellos sería impracticable para el objetivo de este trabajo, dado que se busca analizar los sitios en un contexto de producción, y cualquier sitio con un mínimo tráfico generaría una enorme cantidad de eventos en muy poco tiempo. Esto haría muy difícil cumplir el objetivo de brindar resultados inmediatos, no sólo por el volumen a procesar en sí, sino por la cantidad de procesamiento que es necesario para extraer acciones significativas de mayor abstracción.

Para limitar la cantidad de eventos a procesar en el servidor, y a la vez simplificar este procesamiento con miras a descubrir usability smells, se creó un modelo intermedio de eventos. Estos eventos, llamados *eventos de usabilidad* (o *usability events*), representan acciones más abstractas, con más significado que los eventos de bajo nivel. Esto naturalmente resulta en un menor número eventos a ser procesados, no solamente porque cada uno se compone de múltiples eventos de bajo nivel, sino porque sólo se consideran los eventos de usabilidad que podrían servir para capturar un usability smell, es decir, muchos eventos directamente nunca llegan al servidor.

Un ejemplo de un evento de usabilidad es *Text Input* (carga de texto). Cuando un usuario completa un campo de texto, se generan múltiples eventos de bajo nivel: el click de mouse para seleccionar el campo, los múltiples eventos de teclado para ingresar el texto, incluyendo potencialmente acciones como eliminado de caracteres o pegado desde el portapapeles. Acompañado de estos eventos, hay más información que puede capturarse, como la URL o el tiempo transcurrido. Toda esta información es agregada para generar un único evento *Text Input* que incluye el texto ingresado, el tiempo que llevó ingresarlo, la cantidad de teclas pulsadas, la presencia de una acción de *paste*, la fecha y hora, la URL y el código HTML del campo de texto. Un evento que reúne toda esta información facilita mucho más la búsqueda de problemas que el proceso directo sobre eventos de bajo nivel (aún si éstos incluyen los metadatos descriptos).

4.2. Catálogo de Eventos de Usabilidad

A continuación, se describen todos los eventos de usabilidad definidos en este trabajo. Para cada evento también se definen los valores que funcionan como umbral al momento de la captura.

En el caso de aquellos eventos que afectan un elemento de interfaz particular (por ejemplo, un link), el elemento se almacena, incluyendo la localización en forma de *xpath locator*, código HTML, dimensiones y posición en pantalla, que luego serán usados eventualmente en el reporte para mostrar el elemento afectado. En particular el código HTML se guarda para realizar agrupaciones (clusters) por similitud cuando una familia de elementos está afectada por un mismo usability smell (esto se describirá en el Capítulo 5). Finalmente, todos los eventos se guardan con un *timestamp* del momento de la captura.

A) Tooltip Attempt *Intento de ayuda contextual*

Este evento se genera cuando un usuario posa el mouse sobre un vínculo o una imagen por un determinado período de tiempo. Se asume que el usuario está queriendo obtener información contextual (tooltip) sobre el elemento, como se muestra en la Figura 6.

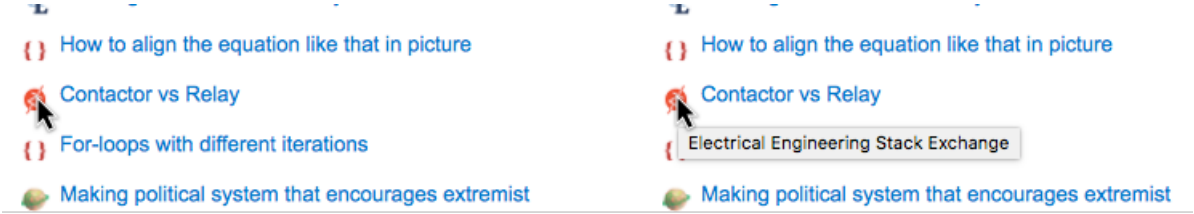

Figura 6 Intento de obtener una ayuda contextual posando el mouse sobre un ícono. A la derecha, la ayuda obtenida en forma de cuadro de texto que describe el significado del ícono.

Más allá de que la ayuda contextual esté definida o no, el sólo hecho de intentar activarla puede advertir una falta de información, o una confusión en la información visible. Por ejemplo, es normal que un vínculo que sólo contiene un ícono como descripción pueda ser sujeto de este tipo de interacción.

Existe un mínimo y un máximo tiempo de espera para este evento. Por defecto están configurados como 1.5 y 5.0 segundos respectivamente. Estos valores fueron obtenidos de la observación directa en experimentos preliminares realizados sobre el

comportamiento de los usuarios. El valor máximo se establece para descartar los casos en los cuales simplemente el usuario dejó de interactuar activamente.

La implementación de este evento en particular es sencilla, sólo hay que interceptar el evento *mousemove* para guardar el momento exacto del último movimiento. Cuando el mouse vuelve a moverse, se captura nuevamente el momento exacto y se calcula la diferencia respecto del último movimiento detectado para conocer el tiempo transcurrido entre ellos. Luego simplemente hay que calcular si este lapso de tiempo está entre los umbrales mínimo y máximo de tiempo de espera (como se dijo anteriormente, 1.5 y 5.0 segundos por defecto). En caso de estarlo, se obtienen los datos del elemento DOM sobre el cual se posó el mouse, junto con el resto de los metadatos comunes a todos los eventos, y finalmente se envía al servidor.

B) Click Attempt *Intento de click*

Este evento se genera cuando un usuario hace click sobre un elemento que no produce ninguna acción. Casos comunes de este problema son textos que parecen ser links por su estilo (ej. subrayado), imágenes que los usuarios desean ampliar cuando no es posible, o el logo del encabezado de la página que muchos usuarios esperan que los lleven a la página principal.

Para detectar este evento, se analiza cada click de mouse realizado sobre algún elemento de la interfaz que no produce una navegación o una acción. Se excluyen los vínculos o los controles de los formularios, ya que se sabe de antemano que van a producir alguna acción. El algoritmo que detecta este evento trabaja principalmente sobre los eventos primitivos de JavaScript *mousedown* y *mouseup*, combinándolos también con la posición del cursor del mouse en ambos momentos, y obteniendo el elemento DOM debajo del cursor. Utilizando esta información, es posible descartar clicks realizados para seleccionar texto o arrastrar elementos. También se descartan clicks hechos sobre elementos de interfaz muy grandes, dado que usualmente esto representa una acción accidental. Por ejemplo, durante pruebas realizadas para validar el algoritmo, se observó que los usuarios a veces hacen click en el navegador sólo para obtener el control de la ventana, también conocido como “foco”.

Un evento Click Attempt almacena el timestamp y la información sobre el elemento afectado.

C) **Flash Scroll** *ráfaga de desplazamiento*

Este evento se genera con una acción rápida de *scroll*, en cualquier dirección y con un largo recorrido, que podría indicar que el usuario ignora los contenidos. Los datos que se capturan son la dirección (URL), posición de la coordenada vertical (***y***) inicial y final, y duración del desplazamiento. Según los experimentos realizados, el umbral que mejor refleja la acción es el alto de la página dividido por 2 segundos.

Existen diferentes motivos por los cuales un usuario podría hacer un *scroll* tan veloz. Por ejemplo, en una evaluación de usabilidad de Aula et al. (Aula, Khan and Guan, 2010) sobre motores de búsqueda, se observa que los usuarios frustrados suelen realizar acciones de scroll hacia arriba y hacia abajo, sin intenciones claras de leer lo que hay en la página de resultados. De forma similar, puede ocurrir que un usuario haga scroll frecuentemente en una sola dirección: hacia arriba, cuando quiere volver al menú de navegación o al encabezado rápidamente, o hacia abajo buscando un contenido específico o el pie de página.

Un evento *Flash Scroll* guarda, además de la información estándar para todos los eventos de usabilidad, las posiciones de inicio y fin del desplazamiento junto con el tiempo total de la acción en milisegundos. Este evento es uno de los pocos que no registra un elemento DOM en particular.

D) **Flash Navigation** *ráfaga de navegación*

Este evento consiste en una secuencia rápida de navegaciones, en la cual un usuario se mueve desde una página hacia otra, para volver repentinamente en un corto lapso de tiempo (la Figura 7 muestra esquemáticamente esta interacción). Esto podría indicar que la página visitada no contiene lo que el usuario espera. En ocasiones, puede suceder que el usuario regrese rápidamente sólo para comprobar que tocó el link indicado, como se detectó en un estudio realizado en 2017 (Vigo and Harper, 2017).

La implementación es relativamente sencilla, pero es uno de los eventos que requiere almacenamiento de información en el navegador (cookies), dado que antes de

terminar de generar toda la información que compone el evento hace falta analizar datos en diferentes páginas. La página considerada de origen es la que contiene el link. Todos los clicks en elementos html de tipo anchor (etiquetas del estilo `<a href="..."></a>`) son capturados, pero sólo se almacena el último junto con el *timestamp*. Los anteriores son ignorados debido a que naturalmente sólo el último habrá producido una navegación, mientras que los anteriores pueden haber sido utilizado para disparar acciones emulando el comportamiento de los botones, lo cual es un uso común de este tipo de elementos. Esta navegación entonces lleva a una página destino, y una vez que se detecta que dicha página está siendo abandonada, lo que es posible gracias al evento *beforeunload* de JavaScript, se puede descubrir cuánto tiempo permaneció el usuario en ella. Luego, en el evento de carga de la siguiente página, se busca si la misma es la página de origen. Si ese es el caso, y además el tiempo de permanencia en la página destino es menor a un umbral determinado, el evento se envía al servidor.

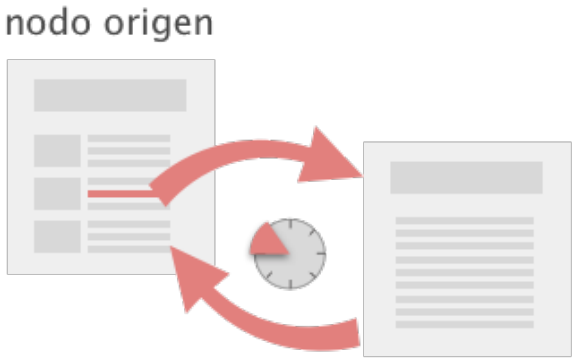

Figura 7 Evento *Flash Navigation*.

El evento almacena el elemento DOM del link empleado, y las dos URLs visitadas (origen y destino), junto con el tiempo de permanencia en el destino. El umbral para el máximo tiempo de espera por defecto en la página de destino es de 3 segundos.

E) Navigation Path *secuencia de navegaciones*

Este evento es similar a *Flash Navigation* en el sentido de que captura una acción rápida de navegaciones, pero en este caso el evento se crea cuando dos o más navegaciones rápidas suceden en cadena, sobre varios nodos (la Figura 8 muestra esquemáticamente

este evento). El camino es capturado, donde cada paso o nodo del mismo contiene el vínculo empleado y la URL de destino. El tiempo de permanencia en cada nodo está también configurado en 3 segundos.

Figura 8 Secuencia de navegaciones.

La implementación tiene muchos puntos en común con la del evento *Flash Navigation*, en la manera de medir el tiempo de permanencia en cada sitio. Cualquier camino de navegación con un origen y al menos 2 nodos más con un tiempo de permanencia por debajo del umbral.

Cada evento contiene la información de los links intermedios utilizados y las URLs de las páginas navegadas. Este es otro de los pocos eventos de usabilidad que no están basados en un único elemento DOM.

F) **Search** *búsqueda*

El evento Search se utiliza para contar el número de búsquedas exitosas o fallidas que los usuarios realizan, si obtuvieron resultados, y si esos resultados fueron de utilidad. El algoritmo que detecta este evento sigue 3 pasos (ilustrados en la Figura 9):

1) Cuando el usuario completa un formulario, el primer paso es descubrir, mediante una heurística sencilla, si se trata de un formulario de búsqueda, analizando propiedades como la leyenda del botón, el número de inputs de texto que contiene, o la presencia de la palabra “search” en diferentes idiomas. Si se determina que es un formulario de búsqueda, el término buscado se almacena en el evento.

2) Luego del envío del formulario, en el segundo paso se examina la página de resultados para hallar una lista de elementos DOM que contenga el

término de búsqueda. Este paso intenta establecer si la búsqueda tuvo éxito o no.

3) Una vez que el usuario abandona la página de resultados, el tercer paso detecta cómo sucedió esto, es decir, si el usuario hizo click en alguno de los resultados (lo que podría indicar que los resultados fueron útiles), o si abandonó la página por otros medios.

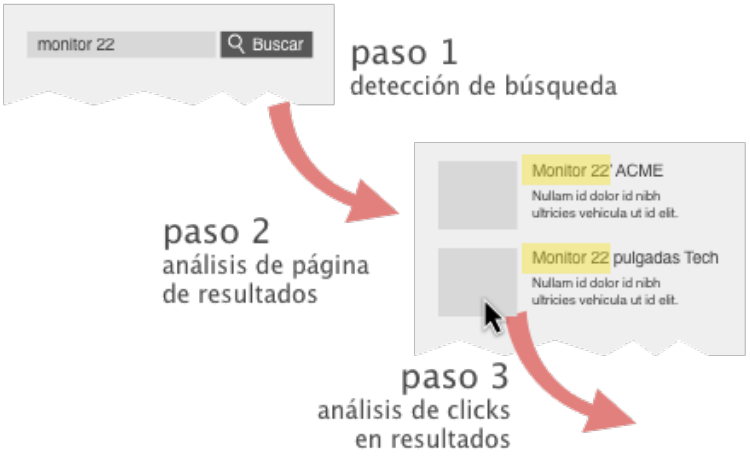

Figura 9 Evento de búsqueda (*Search*).

Al cabo del tercer paso, el evento se envía al servidor con toda la información asociada: detalles sobre formulario, el término de búsqueda, la presencia o ausencia de resultados, y si éstos fueron de utilidad.

Dado que los pasos 1 y 2 están basados en heurísticas, fue necesario ajustarlos mediante experimentación para asegurar que cubrieran la mayor cantidad de casos posibles. Esta validación se explica en detalle en la sección 7.2 del Capítulo 7.

G) Bulk Action *acción en lote*

Este evento se captura cuando un conjunto de elementos en una lista es seleccionado mediante varias casillas de selección de tipo *checkbox*, y luego se envía el formulario con alguna acción. Por ejemplo, si los ítems fueran mensajes, estas acciones podrían ser “eliminar”, “mover a…” o “marcar como leído”.

El algoritmo es considerablemente complejo, en comparación a los anteriores, ya que debe descubrirse primero la presencia de una lista de elementos seleccionables (mediante una casilla de selección o *checkbox*), y esto no siempre está implementado con

etiquetas HTML estándar. Además, detectar la acción es también muy complejo, ya que las maneras de aplicarla son muy variables. Por ejemplo, algunas implementaciones utilizan botones para aplicar la acción (ver el ejemplo de Gmail en la Figura 10), y otras listas de selección. En el último caso, a veces se necesita hacer click en un botón, y otras veces la acción se dispara simplemente al cambiar la selección.

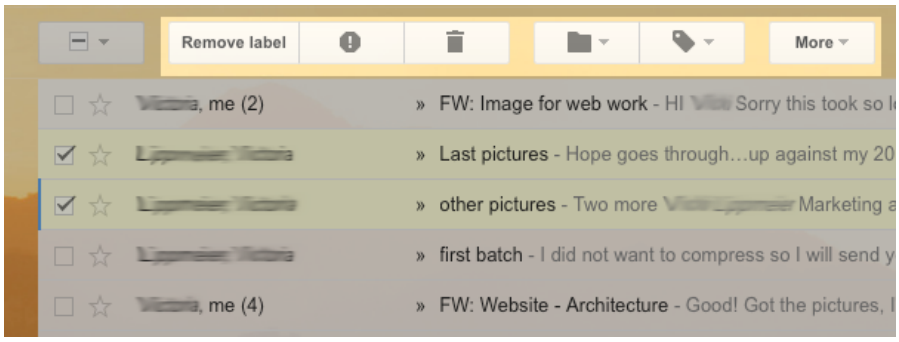

Figura 10 Implementación de Gmail para aplicar acciones en lote.

Teniendo en cuenta las particularidades de este evento, la implementación busca cubrir la mayor cantidad de posibles escenarios utilizando la siguiente heurística:

1) Se detecta una lista de elementos que pueden seleccionarse con un *checkbox*. Para esto, se busca una lista de *checkboxes* alineados horizontalmente, o bien una lista de elementos que tengan forma de *checkbox* con atributos "class" o "role" igual a la palabra "checkbox". El mínimo número de elementos para que se considere una lista es de 4 por defecto.
2) Se detecta una acción en lote. El algoritmo verifica que ninguno de los *checkboxes* esté marcado luego del envío del formulario.

Este evento almacena la información del formulario, junto con la cantidad de elementos marcados.

H) Option Selection *selección de opción*

Este evento detecta una selección realizada sobre una lista de opciones de tipo *select box* o conjunto de *radio buttons*. Particularmente, se encarga de detectar si la opción por defecto de la lista fue alterada. La información que se guarda es, además del elemento DOM (la lista), el índice y el texto de la selección si se trata de un *select box*.

La implementación de este evento consiste en dos partes. La primera parte busca el elemento que está seleccionado al cargar la página, que se guarda como la opción por defecto. Luego se consulta mediante JavaScript el índice seleccionado al momento de enviar el formulario del cual forma parte. En el caso de ser un *select box*, también se guarda el texto de la opción seleccionada. Mediante una simple comparación de los dos índices, se guarda también un valor booleano que indica si la opción por defecto fue alterada o no.

I) **Form Submission** *envío de formulario*

El envío de un formulario es un evento fundamental, dado que permite encontrar múltiples potenciales problemas de usabilidad, al ser ésta la acción más común con alto grado de interacción por parte del usuario.

El aspecto principal capturado por este evento es la estrategia de validación del formulario, es decir, en qué momento luego de que el envío falla se muestran los errores al usuario. El algoritmo que detecta este evento evalúa lo que sucede luego de que el botón de envío es presionado. Si ocurre una navegación, entonces examina la presencia del mismo formulario en la página de destino, para determinar que el envío falló. Este mecanismo exceptúa los formularios de búsqueda, ya que suelen aparecer luego junto con el listado de resultados. Si no sucede una navegación, entonces se asume que el formulario tiene validación del lado del cliente. La estrategia de validación se almacena con el evento, junto con la información del formulario en sí.

El evento también captura los campos del formulario que han quedado en blanco, información que resulta de utilidad para inferir cuáles de estos campos podría ser obligatorios. Con una simple heurística, se puede ver qué campos estuvieron siempre vacíos cuando el formulario fue rechazado, pero nunca lo estuvieron cuando el envío tuvo éxito.

J) **Unfilled Form** *formulario no completado*

Este evento se genera cuando un formulario se completa parcialmente pero no se envía.

En la implementación de la detección de este evento, se observa el evento *focus* de todos los campos de texto, que significa que el cursor de texto se posó sobre alguno

de ellos. Al momento de manejar el evento, se captura el formulario que contiene el campo de texto y se espera al evento *beforeunload* para determinar de qué modo se abandonó la página. Si antes de esto se detecta el evento *submit* del mismo formulario, entonces el evento se descarta, ya que el mismo se considera completado. Notar que en este caso no importa si el envío tuvo éxito o no.

Todo evento *Unfilled Form* guarda el código del formulario y el tiempo que el usuario estuvo completando el formulario antes de abandonarlo finalmente.

K) Text Input *carga de texto*

Cada vez que un usuario completa un campo de texto, se captura la acción en un evento *Text Input*. Este evento recolecta varios detalles de la interacción:

1) **Tiempo** de completado: el tiempo que el usuario tardó en llenar el campo. Técnicamente, es el tiempo que transcurre entre los eventos *focus* y *blur*.
2) Cuenta de ***keystrokes***: es la cantidad de teclas presionadas, que puede diferir del tamaño del texto resultante. Esto puede utilizarse para inferir cierta información adicional sobre el evento. Por ejemplo, si hay muchos menos *keystrokes* que el tamaño final del texto, es posible que haya un autocompletado. Si ocurre lo opuesto, puede ser que el campo sea confuso, lo que lleva a que el usuario borre muchas veces lo que escribió antes de estar seguro.
3) **Texto** resultante: se guarda lo que quedó finalmente escrito, excepto que se trate de tarjetas de crédito o contraseñas.
4) **Corrección**: se captura si el campo contenía texto previamente, antes de ser completado.
5) **Longitudes** en pixels, tanto del texto ingresado como del campo de texto empleado.
6) Información sobre el **elemento DOM**.

Este evento en particular provee información muy útil para ayudar a detectar varios usability smells, dado que el ingreso de texto es una de las interacciones más significativas que realizan los usuarios.

L) Long Request *solicitud larga*

Este evento detecta si una solicitud al servidor (un *request*) lleva mucho tiempo, más que un umbral establecido de 6 segundos. Se busca particularmente un envío de formulario, no cualquier navegación.

Para la implementación de este evento, se debe almacenar una cookie con el *timestamp* al momento de abandonar una página, atendiendo el evento *submit* dado que se trata de un envío de formulario. Luego, en la carga de la página siguiente, se vuelve a tomar el tiempo para calcular el total transcurrido.

El evento guarda el tiempo de demora y las URLs de origen y destino, junto con el formulario que originó el *request*.

4.3. Implementación

Como se explicó al inicio de este capítulo, la implementación de la captura de eventos de usabilidad se realiza en JavaScript. Sin embargo, una vez que los eventos se envían al servidor, se convierten en objetos del modelo que son los que finalmente se utilizan en el proceso de búsqueda de usability smells.

La forma en la que sucede esto es mediante una comunicación entre el módulo JavaScript y el servidor (implementado en Pharo Smalltalk). Para capturar los eventos, el servidor corre un servicio RESTful que recibe los eventos por parte del módulo cliente en formato JSON. Cada envío contiene la siguiente información:

- El **token** de identificación del cliente. Dado que el sistema es en sí un servicio, conviven muchas cuentas de usuario, con lo cual es necesario identificar a cada uno de ellos.
- La **clase** del evento. Esto sirve para que, al momento de recibir el evento del lado del servidor, éste sepa qué clase hay que instanciar.
- Los **datos** particulares del evento. Cada evento de usabilidad siempre tendrá un **timestamp** y la **url** donde se originó el evento. Luego, los eventos que estén basados en un elemento DOM (la mayoría) también guardarán:
  - El **xpath** absoluto del elemento.

- El contenido **HTML** del elemento. El código es automáticamente corregido si hay errores en su codificación, como por ejemplo una etiqueta sin cerrar. Esto ayuda a que el servidor no tenga problemas al intentar procesarlo, evitando problemas de *parsing*.
- Las **dimensiones** y **posición** del elemento dentro de la página.
- La **lista de padres** en el árbol DOM.

Cada evento agregará además su propia información particular. Por ejemplo, *Tooltip Attempt* también almacenará el tiempo de permanencia del cursor en el elemento, en milisegundos.

Toda esta información es enviada vía **Ajax** al *endpoint* del servidor en formato JSON de manera asincrónica, para no estorbar la experiencia del usuario. Cuando el servicio RESTful del lado del servidor recibe un evento de usabilidad, realiza los siguientes pasos para convertir el objeto JSON en un objeto de modelo:

1. Busca la cuenta de usuario que coincida con el token enviado.
2. Crea el evento. Para esto, busca primero la clase que debe instanciar y le envía el mensaje constructor con el resto de los atributos obtenidos del objeto JSON. Cada clase implementará el constructor según los atributos que necesite.
3. Envía el evento recién creado al cliente. Éste se encargará de distribuirlo entre los buscadores de usability smells que lo necesiten para recalcular la presencia de nuevos problemas de usabilidad.

Gracias a la conversión de objetos JSON a objetos de modelo, es posible agregar comportamiento a los eventos, siendo capaces de responder *queries* (consultas) más sofisticadas a la hora de la detección de usability smells.

Capítulo 5. Detección de Usability Smells

En este capítulo se define y describe el corazón de este trabajo: el concepto de usability smell de eventos interacción, especificando las estrategias que permiten su detección automática.

En primer lugar, se provee la definición de usability smell utilizada en este trabajo que se basa en trabajos anteriores, pero se refina para describir específicamente problemas en la interacción real de usuarios en los sitios web, abstrayendo patrones de comportamiento y de estructura de páginas problemáticas, que pueden solucionarse a través de refactorings de usabilidad. En segundo lugar, se describen los lineamientos generales sobre el proceso de detección de usability smells a partir de usability events. Es importante tener en cuenta que el procesamiento que se propone en este trabajo se realiza sobre la marcha, por lo tanto, fue importante aplicar optimizaciones de performance que hagan que la propuesta sea factible. Esto se describe en la tercera sección. A continuación, se

explican detalles técnicos sobre el tratamiento que se da a los usability events a medida que arriban al servidor, lo que resulta en múltiples estrategias de agrupamiento. Finalmente, se provee un catálogo detallado de usability smells de interacción de usuario.

5.1. Definición de usability smells de interacción de usuario

El poder del refactoring está en ayudar a los menos expertos a identificar potenciales problemas en algún aspecto, y a aplicar una serie de pequeños pasos hacia una buena solución para esos problemas. En la jerga del refactoring, estos problemas potenciales se identifican con el término *bad smells.* Los bad smells tienen consecuencias conocidas, y además tienen soluciones conocidas por medio de refactoring (Fowler, 1999). Por ejemplo el refactoring *Extract Method* convierte una porción de código en un método independiente con un nombre apropiado que explica su propósito. Este refactoring sirve para resolver los bad smells *Long Method* ("Método Largo") y *Duplicate Code* ("Código Duplicado").

De forma similar, un *usability smell* es una señal de un diseño de interfaz deficiente, que impide o incomoda al usuario al realizar sus tareas. La detección y corrección de usability smells simplifica el proceso general de evaluación de usabilidad, que los desarrolladores deben realizar de todas maneras mientras trabajan en una aplicación.

Detectar usability smells manualmente puede ser muy demandante, ya que requiere experiencia y tiempo. Además, dependerán del nivel de conocimiento del evaluador, sobre todo si se trata de una evaluación heurística. Por lo tanto, es una tarea que puede ser significativamente acelerada si se aplica algún nivel de automatización en diferentes partes del proceso, que es el motivo por el cual en este trabajo se apunta a la automatización total. Esto haría el proceso práctico y útil para todos los desarrolladores, independientemente de su experiencia en usabilidad. Lograr una detección automatizada de usability smells, permitiría además sugerir refactorings de usabilidad para resolverlos. Sin embargo, hacer posible una metodología automatizada de este tipo, donde los usability smells se puedan ligar directamente con uno o más refactorings concretos, requiere que los catálogos existentes (tanto de refactorings como de usability smells) sean mucho más específicos. Por ejemplo el smell *User Confusion* (confusión del usuario) puede tener múltiples causas como vínculos mal descriptos, falta de información

contextual, o redundancia de contenidos, y por lo tanto, requerir varios refactorings diferentes para solucionarlo del todo, como *Improve the description of process links* (mejorar la descripción de vínculos de proceso), o *Clearly describe errors in executed activities* (describir errores en acciones ejecutadas) (Distante *et al.*, 2014). En consecuencia, para construir herramientas de automatización, es necesario definir usability smells mucho más concretos que los que se encuentran en los catálogos previos.

De esta manera se definen entonces los *usability smells de interacción de usuario*, que son aquellos potenciales problemas de usabilidad que se detectan a partir de analizar el comportamiento de los usuarios con la aplicación. Es decir, los usability smells abstraen cadenas específicas de eventos de más bajo nivel que ya han sido estudiadas, y se conoce que pueden ser síntomas de mala usabilidad. En particular, se hizo foco en aquellos que puedan ser detectados automáticamente, y siempre que sea posible, que puedan resolverse mediante refactoring.

5.2. Detección

Para detectar usability smells, se procesan eventos de usabilidad utilizando diferentes heurísticas. Por ejemplo, si sobre un formulario hay una proporción determinada de eventos *Form Submission* donde se rechazó el envío en el servidor, se detecta el usability smell *No Client Validation*. Este es un ejemplo sencillo, pero las heurísticas pueden ser más complejas y variadas. Incluso pueden consumir más de una clase de eventos.

Cada usability smell cuenta con un buscador específico que implementa la heurística de detección. Estos buscadores llevan el nombre *Usability Smell Finder*. En la implementación de Kobold, existe la clase abstracta con el mismo nombre que incluye comportamiento básico para todos los buscadores, como la creación y la lógica básica para filtrar eventos. También se asigna a los buscadores la responsabilidad de administrar los smells presentes, por lo cual la clase abstracta también implementa el comportamiento necesario para este fin. De la clase `UsabilitySmellFinder` heredan todas las clases que se ocupan de buscar usability smells específicos. Dado que muchos de los smells se aplican a un elemento DOM, existe una clase abstracta intermedia que reúne todo el comportamiento necesario ligado a esta particularidad. De la misma forma, hay otros smells que se aplican sobre una URL, con lo cual también hay una clase intermedia que los reúne. Un diagrama parcial puede observarse en la Figura 11.

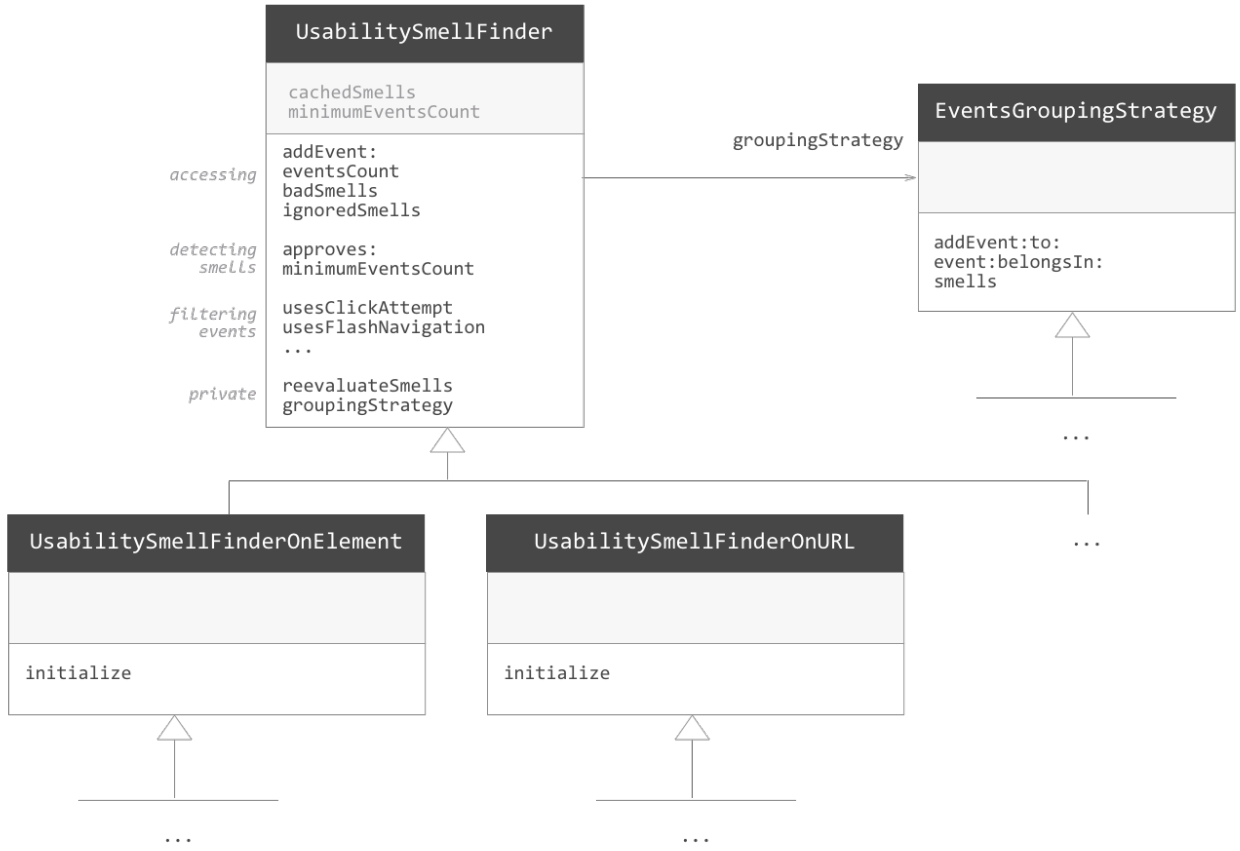

Figura 11 Modelo parcial de buscadores de usability smells.

En la figura puede notarse que ambas clases intermedias, es decir tanto `UsabilitySmellFinderOnElement` como `UsabilitySmellFinderOnURL` sólo redefinen el método `#initialize`. Esto se debe a que la única diferencia entre ellas está en la forma de agrupar los eventos, tal como lo indican sus nombres. Puesto que la organización de eventos está completamente delegada una de las estrategias para este fin, representadas en las subclases de `EventsGroupingStrategy`, basta con definir cuál de estas clases se debe instanciar al momento de la creación del *finder*.

La mecánica básica de todo buscador de usability smells comienza en el arribo de un nuevo evento.

1. Una vez que el servicio encargado de recibir eventos (explicado en la sección 4.1) envía el evento a la cuenta adecuada, se determina quiénes son los *finders* interesados en él. Para esto, cada uno implementa métodos que responden *true* o *false* según el tipo de evento que necesita consumir (ver la categoría *filtering events*). Por ejemplo, si el nuevo evento es un *Click Attempt*, se les pregunta a

todos los buscadores si lo quieren procesar, enviando el mensaje `#usesClickAttempt` a cada uno de ellos.

2. Los buscadores que respondan afirmativamente al método `#uses[TipoDeEvento]` recibirán el mensaje `#addEvent:` con el nuevo evento enviado como parámetro. Siguiendo con el ejemplo, el buscador `UnresponsiveElementFinder` solicitará el evento *Click Attempt* recién llegado.
3. El buscador que recibe el mensaje `#addEvent:`, delega directamente la acción a su estrategia de agrupamiento de eventos. Según la estrategia, los eventos se pueden agrupar según el elemento DOM afectado, la URL afectada, o algún otro criterio ad-hoc. En la sección siguiente se explicará en detalle cómo funcionan estas estrategias.
4. La estrategia de agrupamiento recibe el nuevo evento, lo clasifica y recalcula la presencia del usability smell correspondiente. En el ejemplo del *Click Attempt*, el buscador de smells *Unresponsive Element* clasificará el evento según el elemento DOM afectado, por ejemplo, un encabezado `<H1>`. Si este encabezado recibió un evento que hace que se supere el umbral que dispara la detección, el nuevo usability smell se reporta.

Al momento de recalcular eventos, se busca reprocesar lo mínimo posible, para mantener la velocidad del sistema, que debe dar respuestas inmediatas. Para esto se tomaron algunas decisiones de diseño que se explican en la próxima sección.

5.3. Optimizaciones de performance

Originalmente, la implementación de *Kobold* reprocesaba todos los eventos de todas las cuentas de usuario cada vez que un nuevo evento se capturaba. Esto sirvió para tener respuesta instantánea durante las primeras pruebas, pero al poner en producción con algunos sitios reales, rápidamente se convirtió en un problema al punto de hacer impracticable su uso.

Para entender dónde estaban los puntos que se apropiaban del poder de procesamiento, es importante conocer la organización de toda la maquinaria de búsqueda de usability smells. Como se explicó anteriormente, hay 3 puntos en los cuales se

clasifican los eventos: las cuentas de usuario, los buscadores de usability smells, y finalmente la estrategia de agrupamiento de smells (ver Figura 12).

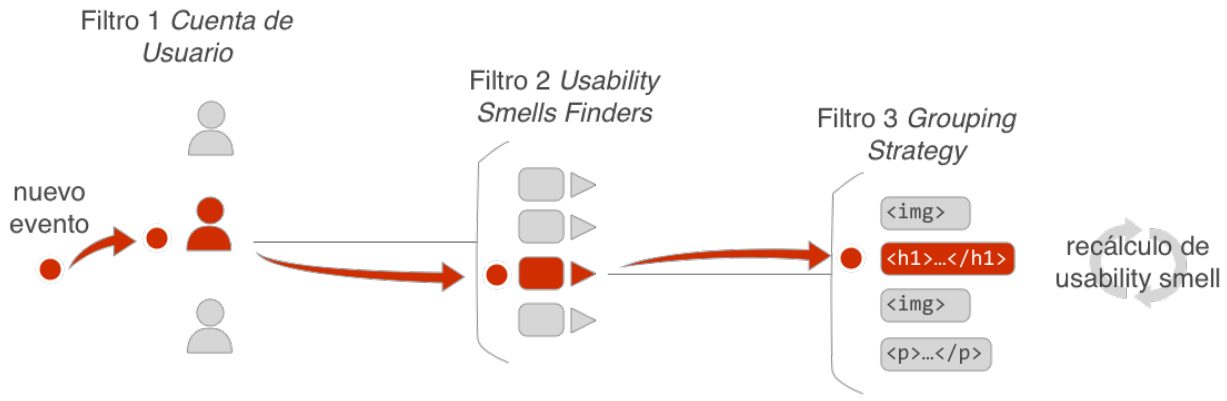

Figura 12 Filtrado de eventos en la detección de usability smells.

Cada filtro es una oportunidad para ahorrar en el número de eventos a procesar. Por ejemplo, observando el primer filtro puede notarse que obviamente no hace falta procesar los eventos de las otras cuentas de usuario, dado que la llegada del nuevo evento no las afectará en absoluto. El segundo filtro, es decir qué buscadores requieren del evento recién llegado, también permite que sólo se reprocese la detección de los usability smells relacionados a dichos buscadores (recordar que un evento puede ser requerido por más de un *finder*). Finalmente, según la estrategia de agrupamiento de eventos, sólo hace falta reprocesar los eventos ligados al elemento que los reúne. En el ejemplo que se ve en la Figura 12, el nuevo evento es un *Click Attempt* sobre un encabezado `<h1>` en particular, con lo cual sólo hace falta procesar los eventos ligados a este elemento junto con el recién llegado, para verificar si el nuevo evento provocó que aparezca un nuevo usability smell *Unresponsive Element*. Tomando estas medidas, el procesamiento de eventos se reduce en aproximadamente 2 órdenes de magnitud.

Pero es posible ir más allá. Aún con la mejora de performance obtenida en los filtrados, el tiempo de procesamiento va a depender igualmente de la cantidad de eventos que se agrupen en el último paso (siguiendo con el ejemplo anterior, en el encabezado `<h1>`). A medida que el número de eventos crezca, el procesamiento será naturalmente más lento. Para evitar incluso esta situación, se realiza una optimización más: toda la información que se pueda sintetizar para evitar recorrer todos los eventos cada vez que

arriba uno nuevo, es almacenada para cada elemento afectado. Puede verse como una implementación muy básica de lo que se realiza en el campo de *Data Streams Mining* (Gaber, Zaslavsky and Krishnaswamy, 2005) que se ocupa de procesar grandes cadenas de eventos en contextos donde la inmediatez del análisis es clave, como en redes de sensores, o tráfico web. En el caso de los *finders*, se guardan datos como porcentajes, cuentas de eventos o promedios. Al ingresar un nuevo evento, simplemente se actualizan estos valores y no se revisitan los eventos previos. Esto hace que la búsqueda de un nuevo usability smell tenga tiempo constante cuando este tipo de síntesis es posible.

Es importante recordar que, si bien en los ejemplos explicados se hace referencia al agrupamiento de eventos en un único elemento DOM, no siempre es el caso. Esto depende de cada estrategia de agrupamiento. La sección siguiente explica en detalle cómo funcionan.

5.4. Estrategias de agrupamiento de eventos

Según el tipo de usability smell que detecta un buscador, necesita procesar los eventos de usabilidad, generalmente agrupándolos por un criterio. Como se mostró en las secciones anteriores, ciertas estrategias de agrupamiento se repiten en muchos buscadores, habiendo dos grupos mayoritarios: los que buscan usability smells en elementos DOM, y los que los buscan en URLs específicas. Dentro de los *finders* que buscan usability smells en elementos DOM, a su vez conviven diferentes maneras de hacerlo. Un modelo parcial de las estrategias puede verse en la Figura 13.

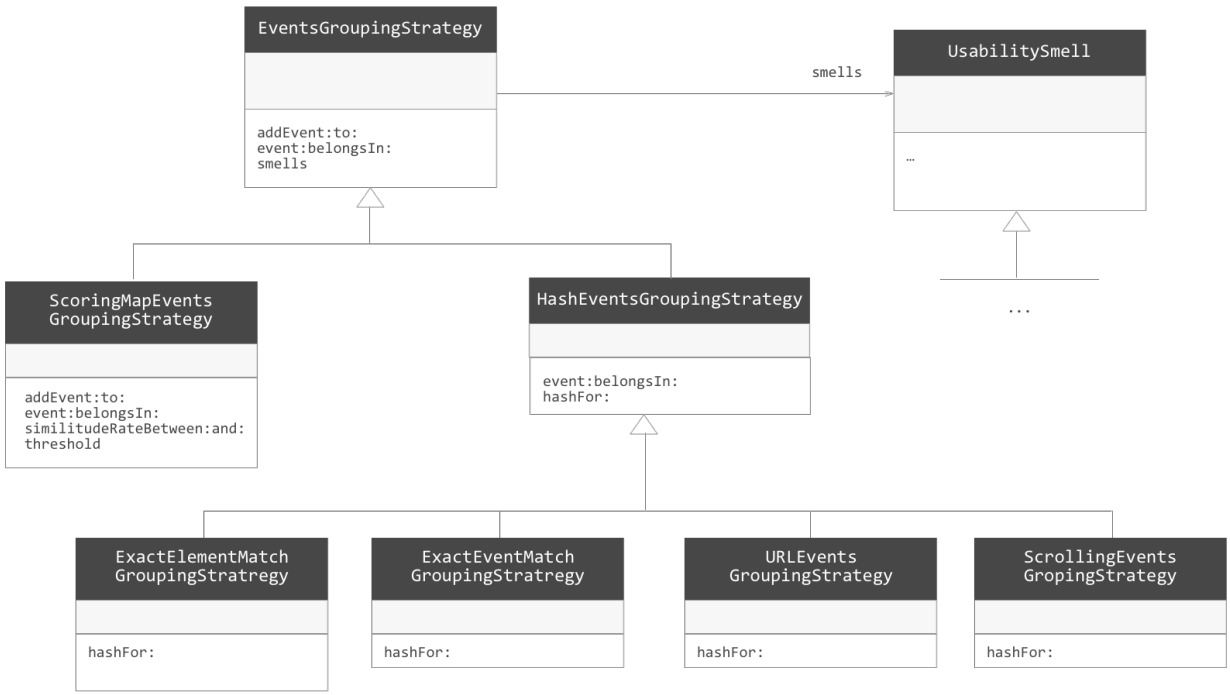

Figura 13 Modelo de estrategias de agrupamiento de eventos de usabilidad.

Como puede verse en el modelo de la figura, existe una clase abstracta que reúne a todas las estrategias, y la mayoría de las estrategias están implementadas como subclases de `HashEventsGroupingStrategy`. Esta clase intermedia define el comportamiento para agrupar eventos según un criterio que se puede representar con una clave de hash. Esto es de gran utilidad por la velocidad de acceso de las estructuras de tipo diccionario o tablas de hash, que en la mayoría de los lenguajes de programación tienen acceso de orden constante (O(1)) o, en el peor caso, orden lineal respecto del tamaño de la clave.

A continuación, se detallarán las estrategias de agrupamiento utilizadas en este trabajo, ya que es una parte importante dentro de la implementación de los *usability smells finders*, en algunos casos llegando a ser determinantes en la lógica de detección.

5.4.1. Coincidencia Exacta de Elementos DOM *`ExactElementMatchGroupingStrategy`*

La manera tal vez más sencilla de verificar que dos elementos DOM sean exactamente iguales consiste en compar su código HTML. Esto permite hallar problemas sobre un mismo elemento que aparece en diferentes URLs, e incluso en distintas posiciones dentro del árbol DOM. La desventaja que tiene está en que, a veces, un elemento puede variar

levemente. Tomemos como ejemplo el menú de navegación del sitio ResearchGate[12] en la Figura 14.

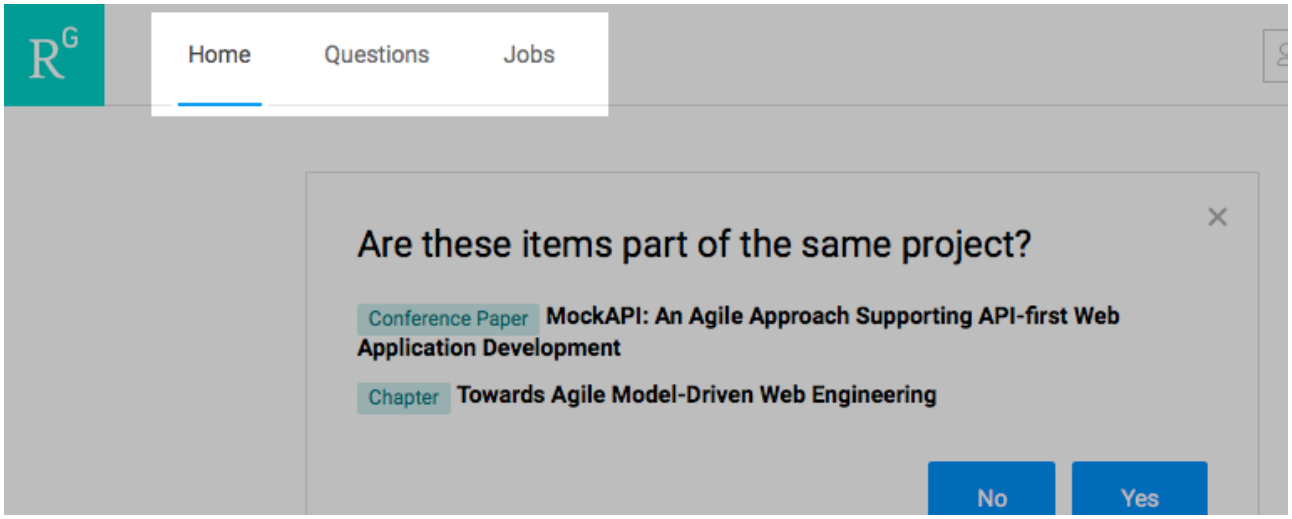

Figura 14 Menú de navegación del sitio ResearchGate.

Cada ítem del menú está representado por el siguiente código HTML:

```
<li class="mainlink mainlink-new "> ... </li>
```

sin embargo, el ítem seleccionado agrega una clase CSS que permite agregar reglas de estilo para destacarlo (en el caso de la figura, la opción “Home”):

```
<li class="mainlink mainlink-new active"> ... </li>
```

Intuitivamente, sería razonable pensar que ambos elementos son en realidad el mismo, más allá del hecho de estar o no seleccionado. En este caso la coincidencia exacta falla, pero igualmente es una estrategia válida que puede ser de utilidad en ciertos contextos. Además, tiene una ventaja muy importante: al ser una comparación exacta, el código HTML puede utilizarse para crear un **hash** que puede utilizarse al momento de agrupar un evento recién llegado. Esto acelera enormemente el proceso de agrupamiento. Esta estrategia de hecho es sólo una de las varias que se han hallado en este trabajo que se basan en funciones de hash para agrupar eventos, ya que es una manera de obtener tiempos constantes (orden *O(1)*) en esta parte tan sensible respecto de la performance.

[12] ResearchGate - https://www.researchgate.net

5.4.2. Similitud Estructural de Elementos DOM `ScoringMapEventsGroupingStrategy`

En algunos casos, sería deseable poder detectar un elemento a pesar de algunos cambios sutiles. En el ejemplo de la Figura 14, podríamos querer que un mismo ítem del menú sea considerado idéntico más allá de estar seleccionado o no, lo que en definitiva produce un cambio en su código HTML. Para conseguir este tipo de detección existen formas de determinar similitud entre dos elementos DOM. Existe otro motivo muy importante para lograr detectar elementos DOM similares: a veces, no sólo se desea detectar un único elemento a pesar de pequeñas mutaciones circunstanciales, sino que se desea detectar **múltiples elementos equivalentes**, dado que todos podrían estar afectados por el mismo usability smell. Consideremos por ejemplo una página de resultados de productos en Ebay (Figura 15).

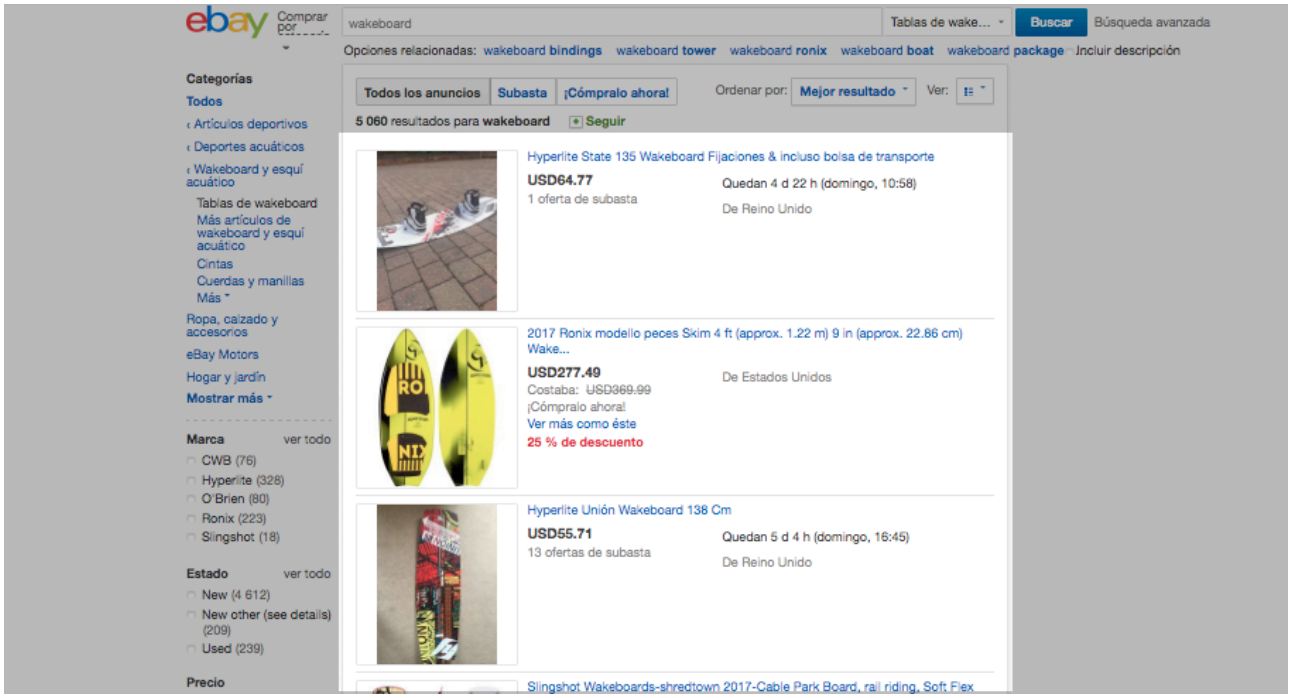

Figura 15 Página de resultados de productos en Ebay.

Todos los productos tienen una foto, un nombre, precio y procedencia, aunque también puede notarse que hay sutiles diferencias: algunos tienen descuentos, otros muestran el tiempo restante de la subasta. Es seguro asumir, a pesar de dichas diferencias, que todos los productos son equivalentes en estructura, y que probablemente surjan de una misma plantilla (con algunos condicionales). Poder detectar esta similitud sería de gran provecho para el trabajo presentado.

El desafío de detectar similitud estructural generalmente se sostiene en el hecho de que la mayoría de los sitios web generan su contenido desde plantillas HTML, completando con información extraída de una base de datos. Por lo tanto, existen varias metodologías para descubrir estas plantillas a partir de las páginas, y producir así un **wrapper**, que podría entenderse como una estructura que funciona como una expresión regular para estructuras, en este caso del modelo DOM. Con un **wrapper**, es posible buscar coincidencias en múltiples elementos para extraer su información pura, es decir, lo que no es parte estructural. Esta metodología es muy común en el campo de *information extraction*.

Para obtener **wrappers**, se suele aplicar un proceso conocido como **wrapper induction**, que implica varios pasos. El primero consiste en agrupar páginas similares en grupos o *clusters*, manual o automáticamente. Luego, se identifican las similitudes estructurales entre ellas para obtener una estructura abstracta que las represente a todas: el **wrapper**. Este proceso resulta de mucho interés para el trabajo aquí presentado, dado que, por un lado, el proceso de generar *clusters* automáticamente es exactamente lo que se busca como estrategia de agrupamiento de eventos. Por otro lado, la generación de un **wrapper** servirá luego en la etapa de refactoring, dado que para aplicar transformaciones a los elementos afectados por un usability smell, primero debe ser posible encontrarlos.

En esta etapa de detección de usability smells, lo importante es lograr detectar elementos DOM similares. Existen varios algoritmos que miden similitud en elementos DOM calculando la distancia de edición entre árboles (Reis *et al.*, 2004; Omer, Ruth and Shahar, 2012). Como estos algoritmos son computacionalmente lentos, se propusieron otros métodos con mejoras de performance, pero sacrificando algo de precisión (Joshi *et al.*, 2003; Buttler, 2004). Además de los problemas con los tiempos de ejecución, precisión y complejidad estructural, la mayoría de las técnicas de extracción de información que utilizan *clustering* son efectivas en páginas completas. Sin embargo, al estar basadas en la comparación interna de los árboles DOM, no obtienen buenos resultados cuando estos árboles son pequeños. En este trabajo, los usability smells suelen afectar elementos DOM pequeños en muchos casos, con lo cual la mayoría de estas técnicas no son efectivas. Existen otros métodos para detectar similitud entre elementos DOM con mejores tiempos de ejecución utilizando *xpath locators*, que marcan la posición de un elemento dentro del árbol DOM (Grigalis and Čenys, 2014). Sin

embargo, estos algoritmos fallan cuando un mismo elemento cambia de posición entre una página y otra, lo cual es frecuente.

Al no encontrar una solución en la literatura que resuelva todos estos problemas, se generó una propia que combina las dos maneras de detectar similitud: estructura interna DOM y direcciones *xpath*. Este algoritmo fue utilizado en un trabajo de grado (Zanotti, 2016) enfocado en el uso de crowdsourcing para mejorar accesibilidad colaborativamente.

El algoritmo, como se dijo anteriormente, se basa en una comparación sopesada de dos aspectos: ubicación en el árbol DOM, y estructura interna. Si la estructura interna es más grande, más peso adquiere en la comparación final, si en cambio la estructura es más pequeña, gana relevancia la ubicación en el modelo DOM representada como una dirección *xpath*. Este método ajustable hace que el algoritmo sea flexible, permitiendo detectar similitud en elementos tanto grandes como pequeños en términos de estructura.

El algoritmo tiene dos pasos: primero genera un **mapa de puntuación** para cada uno de los 2 elemento DOM a comparar. Este mapa captura aspectos estructurales y de ubicación, como se mencionó antes. Luego se comparan ambos mapas, y como resultado se obtiene un **coeficiente de similitud**.

La generación del mapa de puntuación consiste en capturar algunos detalles puntuales de la estructura del elemento DOM. Para cada nodo del elemento se captura: nombre de la **etiqueta** o *tag* HTML, **profundidad** en el árbol DOM, nombres de **atributos** (como por ejemplo clases CSS). Todas estas características conforman la **clave** del mapa, mientras que el **valor** consiste en un número que representa la relevancia que tendrá de esa clave en la comparación final.

Para ver un caso concreto de generación del mapa, tomemos como ejemplo el siguiente código HTML que representa un ítem de una lista:

```
<li id=”comment-1” class=”comment main”>
```

Este código generaría 4 entradas en el mapa: una para la etiqueta **`<li>`** sola, otra para la etiqueta **`<li>`** con el atributo **`id`**, y 2 más que combinan la etiqueta con cada clase CSS (**`comment`** y **`main`**). En este caso, las 4 entradas obtendrían el mismo número de relevancia. Además, si este elemento estuviera repetido dentro de una lista **`<ul>`** (es decir,

si hubiera otros ítems en la lista), sólo uno de ellos se cargaría en el mapa de la lista, dado que no puede haber claves repetidas. Esto es adrede, dado que muchas veces las plantillas HTML tienen elementos repetidos de cantidad variable (por ejemplo, comentarios en un post), y diferentes cantidades de hijos en muchos casos no significa que dos elementos DOM deban ser considerados diferentes. El mapa también contiene una clave por cada ancestro, representadas por niveles de profundidad negativos. Un ejemplo completo de un mapa puede verse en la Figura 16.

Figura 16 Mapa de coeficientes para comparación de elementos DOM.

La figura muestra un mapa de un elemento DOM con la estructura de un post de un blog con varios mensajes. Se puede observar a la derecha el mapa generado, incluyendo las referencias a los ancestros (hasta la etiqueta `<body>`). Notar que los ancestros tienen también un nivel de profundidad (el número antes del carácter @), pero éste es negativo, y decrece a medida que se aleja del nodo raíz. Cada nodo indica, del lado derecho del carácter @, el nombre de la etiqueta y luego el nombre del atributo, con una excepción, en el caso del atributo `class` se genera una clave para cada valor. Esto es así porque dicho atributo es ampliamente utilizado en múltiples frameworks web para identificar elementos. Además, en la etapa posterior de *wrapper induction* se utilizarán las clases CSS para buscar elementos, y este tipo de búsqueda está optimizada en todos los navegadores más populares.

Una vez creados los mapas de los dos elementos a comparar, se obtiene el coeficiente de similitud empleando una fórmula similar al índice de Jaccard para calcular similitud de conjuntos. La fórmula para obtener el coeficiente de similitud entre dos mapas *m* y *n* es la siguiente:

$$S(m,n) = \frac{\sum_{k \in (K(m) \cap K(n))} \max(m[k], n[k]) * 2}{\sum_{k \in K(m)} m[k] + \sum_{k \in K(n)} n[k]}$$

donde *K(m)* es el conjunto de claves del mapa *m*, y *m[k]* es la relevancia para la clave *k* en el mapa *m*. La sumatoria en el diviendo obtiene la intersección de las claves de ambos mapas. Para cada una de estas claves, se obtiene la relevancia en ambos mapas y se toma el mayor valor (con la función *max(m[k], n[k])*) multiplicado por 2, priorizando la mayor relevancia. El término del divisor suma la relevancia total de ambos mapas.

El algoritmo tiene orden *O(n)*, es decir lineal con respecto al número de nodos de los árboles comparados. Si bien tiene dos pasos, uno para construir los mapas y otro para compararlos, ambos son lineales. Debe notarse que calcular la intersección de claves en el segundo paso también es un proceso de orden lineal, ya que sólo una de las estructuras se recorre mientras que la otra se accede por clave, lo que para mapas y diccionarios generalmente se considera *O(1)*, es decir acceso constante (si bien en algunos lenguajes de programación se considera lineal con respecto al tamaño de la clave, como se explicó en la descripción de las estrategias basadas en claves de hash). En la sección 7.5 se describe un experimento realizado para validar esta estrategia, en comparación con otros algoritmos conocidos de similitud de elementos DOM y *clustering*.

El algoritmo descrito está implementado en la estrategia de agrupamiento `ScoringMapEventsGroupingStrategy`.

5.4.3. Otras Estrategias

Además de la estrategia de coincidencia exacta para elementos DOM, existen otras que fueron creadas para *finders* específicos, todas ellas basadas en claves de hash. Por ejemplo, los buscadores de usability smells que afectan una URL particular y no un elemento DOM, agruparán los eventos con el criterio de la URL afectada, para luego informar el smell sobre ella.

Cada una de las estrategias restantes será explicada en el catálogo incluido en la próxima sección, ya que cada una forma parte de la lógica de detección de un *finder* en particular, y sólo cambia el criterio para generar la clave de hash, mecanismo que ya ha sido explicado en la estrategia `ExactElementMatchGroupingStrategy`.

5.5. Catálogo de Usability Smells

A continuación, se presenta un catálogo de usability smells. Cada uno se describe con un nombre, descripción y un ejemplo. Luego en la Tabla 1 se conecta cada smell con los eventos de usabilidad que se necesita procesar su detección.

1. **Undescriptive Element** *elemento no descriptivo*

Todo componente interactivo de una interfaz, ya sea un botón, un link u otro tipo de control, necesita de un significante claro, es decir alguna pista que muestre al usuario lo que se puede hacer con él (Norman, 2013). Cuando esto no sucede, los usuarios pueden no encontrar el control que necesitan para completar una tarea. A veces, sin embargo, puede suceder lo opuesto: un usuario puede encontrarse con un elemento que no puede comprender para qué sirve en un primer vistazo. En estos casos, un comportamiento normal para investigar su propósito consiste posar el mouse sobre el control, dejarlo quieto unos instantes, y esperar que aparezca un *tooltip*.

Cuando muchos usuarios intentan obtener una ayuda tipo *tooltip* de un elemento de la interfaz, esto puede indicar que el elemento no es suficientemente auto-descriptivo. Según las guías de diseño para desarrolladores de Windows 10: "*Una ayuda valiosa ayudará a aclarar una acción poco clara*".

Como ejemplo, la Figura 17 muestra una captura de Google Drive donde se encontró que los usuarios posaban el mouse sobre sus nombres (en el caso de la figura, "Alejandra") buscando una opción de *log out* para salir del sistema. Los nombres de usuario fueron eliminados de este encabezado en versiones posteriores de Google Drive.

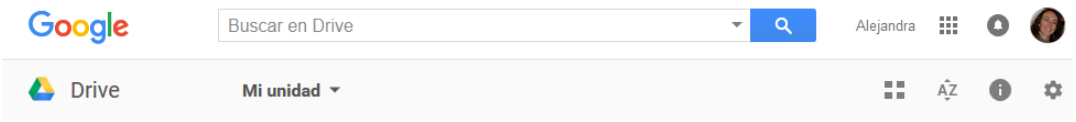

Figura 17. Captura del encabezado de Google Drive en mayo de 2016

La implementación de la detección de este smell consiste en procesar los eventos *Tooltip Attempt* (ver capítulo 4.A). Cada vez que un nuevo evento de este tipo arriba al servidor, se lo agrupa junto con los otros eventos que afectan al mismo elemento DOM, en caso de no estar, se carga el elemento. Luego se recuentan los eventos para el elemento afectado, si superan un umbral en una ventana de tiempo, se detecta el smell.

Una característica interesante de este smell es que aparecerá sin importar si el *tooltip* está realmente implementado o no, dado que la simple necesidad del usuario de mostrarlo puede indicar un problema con la descripción del elemento de todas maneras. Otra característica interesante que tiene que ver con la captura del elemento, es que también detecta sin haberlo buscado, otro comportamiento de los usuarios que están explorando. En las pruebas preliminares con voluntarios avanzados en el uso de navegadores se observó que, cuando un link no es claro, dejan el mouse posado sobre el mismo, pero para revisar la barra de estado y ver la URL de destino. Esto naturalmente dispara la captura de un *Tooltip Attempt*, que, si bien estrictamente sería un falso positivo, sirve de todas maneras para detectar el mismo síntoma.

2. **Misleading Link** *vínculo confuso*

Ante la falta de descripción de un link, los usuarios pueden buscar un *tooltip* que los ayude a aclarar adónde los llevaría, pero se ha observado otro comportamiento más práctico para averiguar esto: presionar el link. Los usuarios que optan por esta opción generalmente dan un rápido vistazo y retornan a la página de origen.

El smell *Misleading Link* es similar a Undescriptive Element pero específico para links. Además de los *Tooltip Attempts*, este smell captura eventos *Flash Navigation*. Recordemos que estos eventos implican una navegación muy rápida de ida y vuelta, que, de reiterarse, puede interpretarse como un arrepentimiento al ver que el contenido buscado no está presente. Puede ser también que simplemente se repita un patrón de comportamiento que consiste en explorar la página de destino como se explicó anteriormente. En cualquier caso, la descripción del link puede ser el problema.

Este problema ha sido documentado en la motivación para aplicar el patrón “Descriptive, Longer Link Names” (nombres de vínculos más largos y descriptivos)(van Duyne, Landay and Hong, 2006) que marca la frustración de los usuarios que continuamente van hacia delante y hacia atrás al navegar por vínculos que no les interesan. Un ejemplo común de un vínculo pobremente descripto es el usual “click aquí”, que puede llevar a un contenido que el usuario no espera.

3. **No Processing Page** *ausencia de página de proceso*

La falta de *feedback* produce incertidumbre en los usuarios, en particular la falta mensajes que informan sobre procesos largos. En el contexto de las aplicaciones web, cuando un usuario presiona un botón o un link, espera una respuesta con cierta inmediatez. ¿Cuánto tiempo esperamos antes de que nuestra mente perciba que la reacción es producto del evento? Según la siguiente clasificación extraída del libro "Usability Engineering" de Nielsen (Nielsen, 1994) existen 3 niveles de demora característicos:

- **0.1 segundo** es el límite aproximado para que el usuario sienta que el sistema está reaccionando instantáneamente, es decir que no hace falta mostrar un mensaje al respecto, sólo el resultado.
- **1.0 segundo** es el límite aproximado para que **flujo de pensamiento del usuario** no se interrumpa, incluso si el retraso es percibido. Normalmente, no se necesita un mensaje que indique que hay un proceso ejecutándose cuando el retraso está entre 0.1 y 1.0 segundos, pero el usuario sí pierde la sensación de estar operando directamente sobre los datos.
- **10 segundos** es el límite aproximado para **mantener la atención del usuario** enfocada en el diálogo. Para demoras mayores a este límite, lo usuarios desearán hacer otra cosa mientras esperan que la computadora termine, por lo cual habría que mostrarles un mensaje que indique para cuándo se estima que el proceso estará finalizado. Un mensaje de este tipo durante la demora es particularmente importante si el tiempo de respuesta tiene a ser muy variable, ya que los usuarios no sabrán qué esperar en ese caso.

Este no es el único estudio sobre los umbrales de demora y pérdida de atención del usuario, y lo cierto es que varían de un estudio a otro, pero típicamente se encuentran entre de 8 a 16 segundos (Nah, 2004). En los experimentos realizados en este trabajo se han obtenido mejores resultados con el umbral sugerido por Nielsen, es decir de **10 segundos**. Muchos sitios y aplicaciones web que realizan procesamientos demandantes en tiempo utilizan una página de espera donde muestran claramente que se está procesando un pedido, y en ocasiones agregan una barra de progreso (ver en la Figura

18 la página de proceso que el sitio despegar.com muestra al usuario durante el proceso de búsqueda de vuelos).

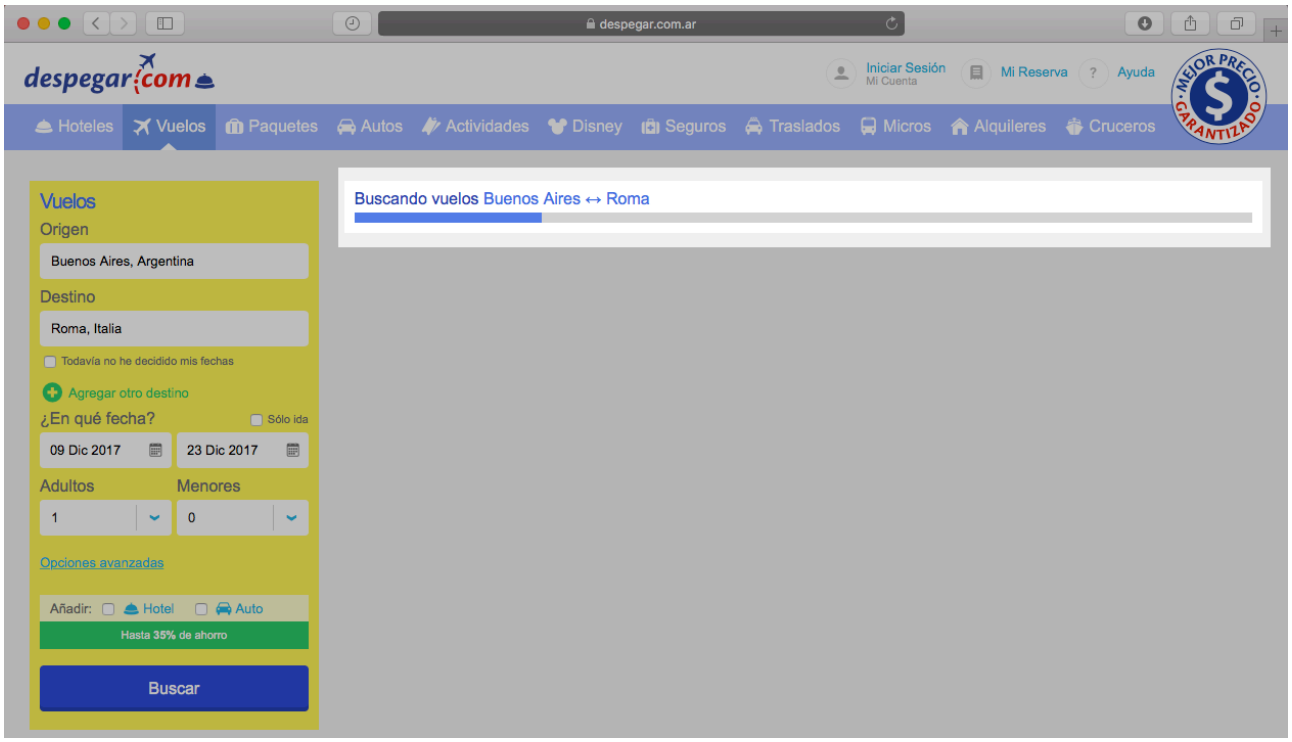

Figura 18. Página de proceso del sitio despegar.com. Se puede observar una barra de progreso mientras se buscan opciones para una búsqueda de vuelo.

La mecánica de la detección del buscador consiste en medir el tiempo de demora promedio para acceder a una URL determinada. Si este promedio supera el umbral de 10 segundos, se detecta el smell. Este es un buen ejemplo de *finder* que no afecta a un DOM element, sino a una URL, y por lo tanto utiliza una estrategia de agrupamiento de eventos con estas características. El buscador en sí es subclase de `UsabilitySmellFinderOnURL`. Es también un buen ejemplo sobre la síntesis de datos que se describió en la sección 5.3, ya que se almacena el promedio de demora y el número de eventos. Cuando un nuevo evento arriba, se incrementa el número almacenado de eventos y con esto es posible actualizar el promedio almacenado mediante un simple cálculo.

4. **Free Input For Limited Values** *campo de texto libre para valores limitados*

Este smell se detecta cuando un campo de texto libre se utiliza para ingresar datos de un rango limitado de valores, como ciudades o países. Los usuarios se ven obligados a escribir el texto completo, cuando en realidad las opciones son restringidas. En el caso en

el que hay unas pocas opciones, un *select* es más apropiado que un campo de texto, y cuando el número de opciones es más bajo, entre 5 y 7 valores, un conjunto de *radio buttons* es más apropiado.

Este smell puede aparecer también en el caso en que las opciones no estén en rigor restringidas, pero aun así haya un conjunto pequeño de opciones muy populares. En ese caso, un autocompletado debería ayudar a los usuarios a escribir menos texto, y además hacerlos sentir más seguros de lo que están ingresando al verlo entre las opciones sugeridas.

La mecánica de detección consiste en procesar los eventos *Text Input*, que proveen información detallada sobre el completado de un campo de texto por parte de un usuario. De toda la información que se obtiene de estos eventos, en particular lo interesante es el texto que se ingresa. El primer paso consiste en agrupar los eventos por elemento DOM, en este caso el campo de texto afectado. Para cada campo de texto, se evalúan los textos ingresados. Es importante recordar que se exceptúan de este análisis los datos sensibles como tarjetas de crédito o contraseñas. El análisis que se hace sobre los textos es el siguiente:

1. Se agrupan los valores por similitud. Por defecto, el algoritmo empleado para esto es la distancia de edición de Levenshtein (Levenshtein, 1966), pero es posible alterarlo.
2. Para cada grupo, se mide la proporción que representa en el total de los valores. Los valores marginales que no están en ningún grupo mayoritario se agregan a un grupo especial llamado "*Other*".
3. Si el número de grupos mayoritarios está dentro de un rango especificado, y el grupo de valores marginales es suficientemente bajo, se detecta el smell.

De este mecanismo se desprenden varios valores de corte, que pueden configurarse, aunque tienen un número por defecto, obtenido de los múltiples análisis realizados con aplicaciones reales.

- **Número mínimo de eventos**: *¿Cuántos eventos Text Input deben afectar a un campo de texto antes de considerar el análisis?* Este número está puesto por defecto en 200 eventos, lo cual es relativamente alto en comparación con otros

buscadores, pero es importante contar con una buena base de valores antes de considerar el smell *Free Input for Limited Values*, ya que en caso contrario podría detectarse tempranamente con valores como nombres de personas o códigos postales, donde no tiene sentido.

- **Proporción mínima de grupos**: ¿Cuál es el tamaño mínimo para los grupos de valores similares, para considerarlo un valor frecuentemente ingresado, y no un valor marginal? Este valor es importante para la detección porque determina qué es un valor frecuente, y en consecuencia define el número máximo de valores frecuentes que puede haber. Está fijado por defecto en 5%, lo cual da un rango de valores no muy amplio, de entre 18 y 20 dependiendo del tamaño del grupo marginal.
- **Proporción máxima de valores marginales**: *¿Qué tamaño máximo puede tener el grupo de valores marginales, en proporción?* El valor por defecto máximo es del 10%. Un número mayor implicaría que la cantidad de valores libres es suficiente para
- **Algoritmo de similitud**: ¿Cuál es el criterio para considerar que dos valores levemente distintos deben considerarse en realidad el mismo?

Todos los valores son ajustables. Si bien los valores por defecto han sido validados en múltiples experimentos, hace falta más investigación para determinar cuáles son los óptimos ante diferentes flujos de tráfico.

La herramienta Kobold reporta este usability smell con una visualización de las proporciones de valores y una tabla con los valores en sí, junto con las ocurrencias de cada uno (ver captura en la Figura 19).

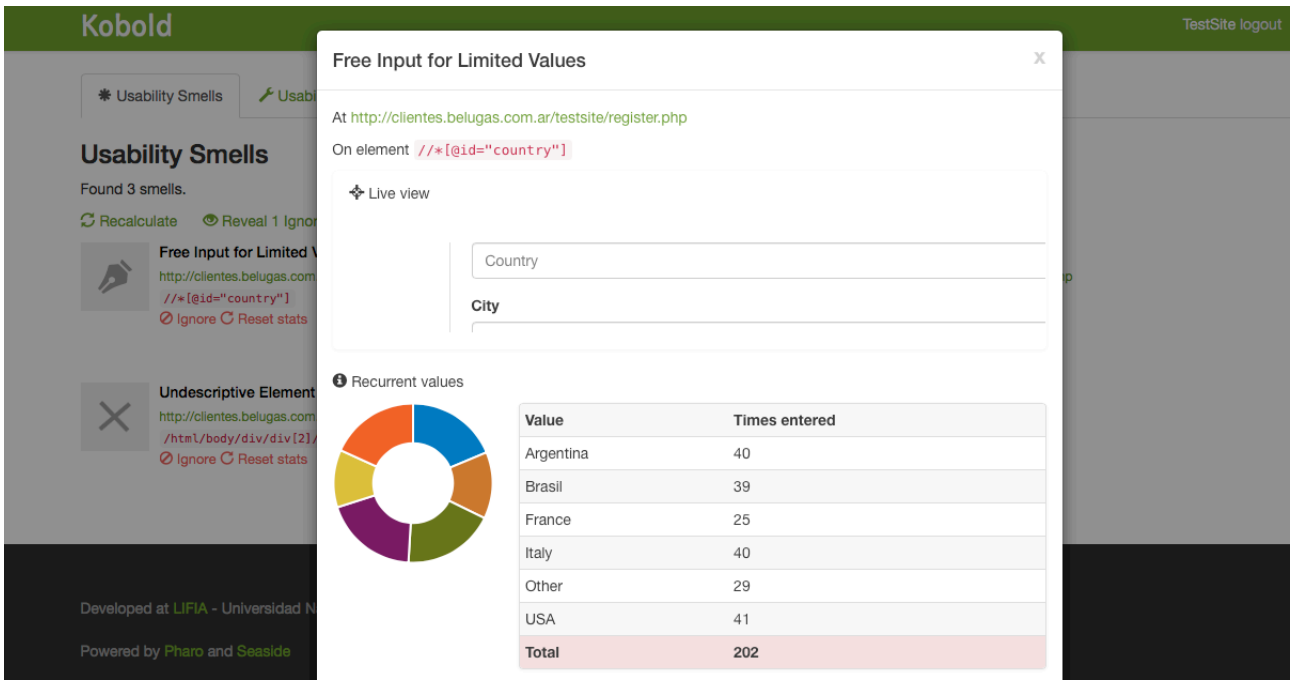

Figura 19. Reporte de un usability smell *Free Input for Limited Values* sobre un campo de texto en el que se debe ingresar un país.

En la captura de pantalla también se ve que el reporte indica cuál es el elemento afectado mediante su *xpath* y una vista en vivo del mismo dentro de un elemento `<iframe>`, junto con una URL donde se lo puede encontrar (ya que puede estar en más de una).

5. **Unformatted Input** *campo sin formato*

Muchos campos de texto que completan los usuarios sirven para ingresar datos que responden a un formato determinado, como fechas, números de teléfono o códigos postales. En estos casos, el formulario debería ayudar al usuario a darle formato al dato ingresado. Sin embargo, en muchos casos se ofrece un campo de texto libre, forzando al usuario a determinar cuál es el formato correcto. Al ingresar números de teléfono, por ejemplo, el formato automático puede ser de gran utilidad, especialmente cuando el sitio tiene una audiencia internacional (*"¿debo ingresar '+' antes del código de país? ¿Está permitido?"*). En las experiencias durante la investigación para este trabajo, por ejemplo, se halló que empleado calendarios en vez de campos de texto para fechas puede mejorar la satisfacción en el uso, en el contexto de sitios de e-commerce (Grigera *et al.*, 2016).

Un smell *Unformatted Input* detecta las situaciones donde se utiliza un campo de texto libre para ingresar datos que deben tener un formato específico. Para detectar este usability smell, se procesan eventos *Text Input* para cada campo de texto, y se comparan

sus valores utilizando diferentes expresiones regulares (una por formato). Cuando un porcentaje de valores que coinciden con una única expresión regular supera un umbral determinado, se detecta el smell.

Los parámetros configurables del buscador de *Unformatted Input* son 2: por un lado, las **expresiones regulares** (por defecto fechas, números y email, pero fácilmente pueden agregarse más). El otro parámetro es la **proporción mínima de valores** que deben responder a un formato (por defecto 0.7).

6. Unmatched Input Size *tamaño de campo inadecuado*

No es infrecuentemente encontrarse con campos de texto que tienen un tamaño muy diferente de los valores para los cuales está diseñado, ya sea más corto (en cuyo caso se oculta parte del texto ingresado) o más largo. Este problema trae confusión al usuario, como lo indican autores como Wrobleski (Wroblewski, 2008).

Un tamaño adecuado ayuda al usuario a prever qué tipo de dato se espera completar. La Figura 20 muestra un ejemplo de cómo se ve un formulario donde todos los campos tienen una longitud uniforme, en contraste con otra versión del mismo formulario donde se han adecuado los tamaños de los campos de texto a la longitud típica de los datos que se esperan.

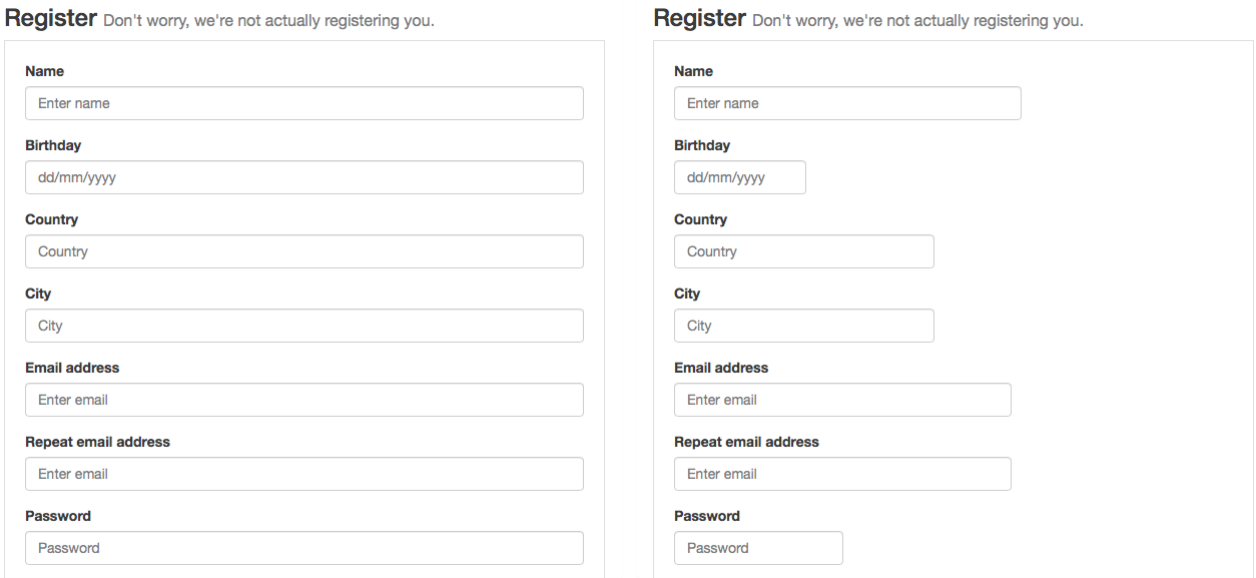

Figura 20. A la izquierda, un formulario con los campos de texto de longitud uniforme. A la derecha, el mismo formulario con los tamaños ajustados, dando una mejor idea del tipo de dato se espera en cada uno.

Para detectar este usability smell, se analizan los eventos *Text Input* sobre un campo particular. Medir el tamaño del texto en pixels es la parte más crítica, dado que depende de factores como el tipo de fuente, o de cada caracter en sí, puesto que varían en ancho en la mayoría de las tipografías. Se calcula entonces un tamaño aproximado reproduciendo el *rendering* del texto tal cual lo procesa el navegador, pero en un *canvas* HTML5 oculto. Esto se mide contra el tamaño en pixels del campo de texto para determinar la diferencia. Cuando el promedio de longitud de los textos tiene una diferencia substancial de tamaño con el campo, se detecta el smell. Tanto el tamaño de texto promedio como la varianza son almacenados en *cache* y recalculados de forma incremental con la llegada de cada nuevo evento usando técnicas de Data Stream Mining.

Los parámetros ajustables son dos: por un lado, la diferencia de tamaño, por defecto fijada en -60 pixels para considerar un campo excesivamente largo, y 40 pixels para considerarlo demasiado corto. La distinción entre ambos valores se hace porque un campo demasiado corto es más problemático que uno demasiado largo, ya que hace que el texto excedido se oculte (por ende, la tolerancia más baja de 40 pixels).

7. **Forced Bulk Action** *acción en lote forzada*

Es común que las aplicaciones web muestren una lista de ítems (como productos o mensajes) y ofrecer acciones en lote sobre ellos. Los usuarios típicamente realizan estas acciones seleccionando primero un grupo de ítems utilizando *checkboxes*, luego seleccionando la acción, por ejemplo “Eliminar” o “Mover a…”, para finalmente aplicar dicha acción con un botón, o en ocasiones sólo seleccionando la acción se aplica automáticamente sin necesidad de confirmación (ver Figura 21).

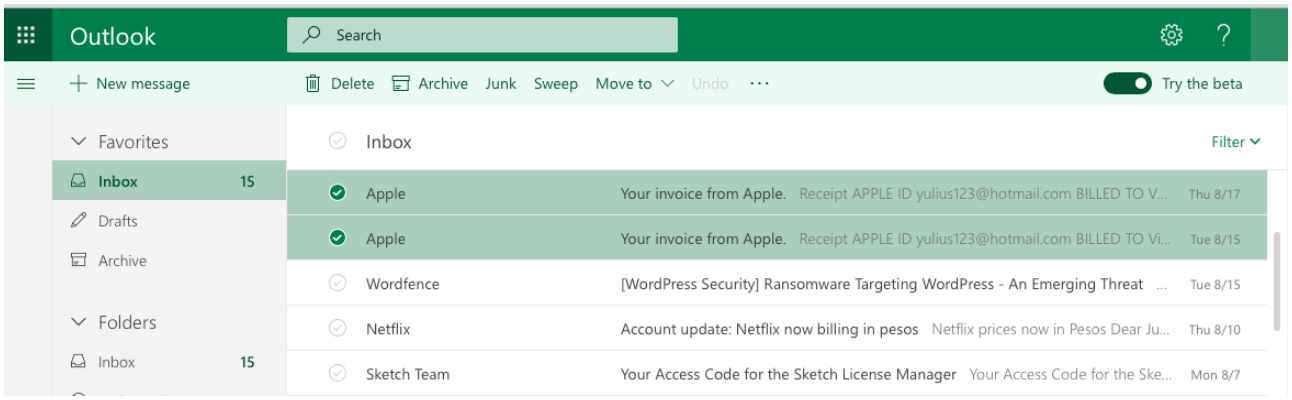

Figura 21. Interfaz de Outlook para manipular múltiples mensajes de una sola vez. Arriba pueden verse las acciones como “Delete”, “Archive”, “Junk”, y debajo, el método de selección de correos.

Aunque este estilo de interacción funciona bien cuando se quiere afectar a un grupo de ítems a la vez, lleva demasiados pasos si lo que se quiere es realizar la acción sobre uno solo. En los casos en que esto último sea lo más frecuente, y los usuarios no tienen (o no ven) otra alternativa más rápida, el smell *Forced Bulk Action* se detecta.

La mecánica de detección es relativamente simple, ya que los eventos que el buscador consume tienen la información necesaria ya procesada. Cada evento *Bulk Action* contiene el número de elementos que fueron afectados por una acción en lote. La heurística consiste entonces en contar cuántos de ellos se aplicaron sobre un único elemento. Si la proporción de éstos sobre el total de eventos es igual o mayor que 0.8 (por defecto), el smell se detecta.

Este smell surge del trabajo previo en refactorings de accesibilidad, y aparece en el catálogo como “Difficult interaction to apply operations on elements” (acción difícil de aplicar sobre un elemento) (Garrido, Firmenich, *et al.*, 2013). Para usuarios no-videntes, la presencia de *Forced Bulk Action* es un problema bastante serio, ya que requiere seguir una cantidad de pasos que implica ir hacia delante y hacia atrás múltiples veces con un lector de pantalla, primero para leer la lista de elementos y seleccionarlos (la casilla suele estar *antes* del contenido que hay que leer para saber cuál es el elemento, lo cual lo vuelve aún más incómodo), para luego volver a buscar la operación, que si bien en ocasiones se repiten al final del listado, suelen estar en el encabezado del mismo.

8. **Overlooked Content** *contenido ignorado*

Ciertos contenidos suelen ser rápidamente salteados en acciones rápidas de *scroll*, con lo cual es probable que los usuarios no los vean. Además, si usualmente se detienen en la misma sección de la página (posición *y*), es muy probable que el contenido alcanzado sea más fácil de encontrar si estuviera más cerca del tope. Si bien hoy en día los usuarios tienen más voluntad de hacer *scroll* (abundan hoy en día los diseños tipo *parallax* con animaciones que se van activando con el *scrolling*), siguen necesitando un buen motivo para hacerlo, ya que agrega una acción de interacción de todos modos.

La manera de detectar este usability smell es a través del análisis de eventos de usabilidad *Flash Scroll*. El *finder* que busca el smell *Overlooked Content* tiene su propia estrategia de agrupamiento de eventos (representada en la clase

`ScrollingEventsGropingStrategy`), y es allí donde se concentra la mayor parte de la lógica. Para cada URL afectada con eventos *Flash Scroll*, se forman grupos de afinidad según la coordenada de inicio y la coordenada de fin, de modo que los eventos que representan acciones de *scroll* equivalentes (con un número de pixels de tolerancia) se colocan en el mismo grupo. Luego, simplemente se cuentan los eventos de cada grupo, cuando un grupo supera un umbral de eventos en una ventana de tiempo, el smell *Overlooked Content* se detecta.

Además de indicar un potencial problema de usabilidad, este smell provee información valiosa en el caso de que el contenido ignorado sea considerado importante por el dueño de la aplicación. Ciertas herramientas de análisis proveen mapas de scroll en los cuales se ve cuán lejos del tope los usuarios *scrollean*, pero no siempre muestran la velocidad.

9. Distant Content *contenido distante*

El usability smell *Distant Content* detecta patrones de navegación reiterados, en los cuales los usuarios sólo permanecen en los nodos intermedios por un breve instante. La lógica detrás de este smell radica en que los usuarios están buscando los contenidos en el nodo final, pero esto requiere un camino largo que recorrer. Una forma usual de estudiar usabilidad en la navegación es comparar un camino óptimo con caminos reales seguidos por usuarios, ya sea en un contexto real o en un experimento de laboratorio (Atterer, Wnuk and Schmidt, 2006). El smell Distant Content, sin embargo, indica que el camino seguido desde el nodo inicial hasta el final puede ser demasiado largo, donde un solo nodo extra puede considerarse demasiado largo.

El *finder* encargado de detectar este smell es el único que es subclase directa de la clase abstracta `UsabilitySmellFinder`, es decir, no basado en hallar un smell sobre un elemento DOM ni sobre una URL. De hecho, el smell abarca muchas URLs. Al igual que en *Overlooked Content*, la lógica de detección está mayormente implementada en la estrategia de agrupamiento de eventos, ya que es donde se generan los conjuntos de eventos *Navigation Path*. La estrategia (implementada en la clase `PathEventsGroupingStrategy`, perteneciente al grupo de estrategias basadas en hashing) se ocupa de

10. **No Client Validation** *ausencia de validación en el cliente*

Reportar errores en formularios web a tiempo y de forma clara es una práctica de usabilidad ampliamente recomendada por muchos autores (Wroblewski, 2008; Seckler *et al.*, 2014). Según Don Norman explica en su libro *The Design of Everyday Things* (Norman, 2013):

"Los diseñadores deberían buscar su atención casos en los cuales las cosas van mal, no sólo cuando las cosas funcionan como se planeó. De hecho, es allí donde puede surgir la mayor satisfacción: cuando algo va mal, pero la máquina resalta los problemas, entonces la persona entiende el error, tomar las acciones necesarias, y el problema se resuelve."

"Designers need to focus their attention on the cases where things go wrong, not just on when things work as planned. Actually, this is where the most satisfaction can arise: when something goes wrong but the machine highlights the problems, then the person understands the issue, takes the proper actions, and the problem is solved"

Este smell en particular detecta formularios que tienen una alta tasa de rechazo, donde la validación sucede en el servidor, es decir, luego de que el navegador envió el formulario y se recargó la página. Este escenario se detecta analizando eventos *Form Submission* que son capaces de informar si un formulario se envió, pero luego volvió a aparecer.

La detección de *No Client Validation* se basa en calcular una proporción entre envíos de formulario rechazados y aceptados. Por defecto esta proporción está fijada en 0.4. Una característica especial de este usability smell, es que es capaz de inferir qué campos del formulario afectado podrían ser obligatorios, como se muestra en el siguiente código (recordar que los eventos *Form Submission* almacenan los campos de texto vacíos en cada envío):

```
requiredInputs
  | emptyInputsWhenFailed emptyInputsWhenSucceeded |
  emptyInputsWhenFailed := self failedSubmission flatCollect: #emptyInputs.
  emptyInputsWhenSucceeded := self successfulSubmissions flatCollect: #emptyInputs.
    ^(emptyInputsWhenFailed copyWithoutAll: emptyInputsWhenSucceeded) asSet
```

el método `#requiredInputs` obtiene dos grupos de campos de texto. Por un lado, un primer grupo que contiene todos los campos que estuvieron vacíos en al menos un envío fallido (la variable temporal **`emptyInputsWhenFailed`**), y por el otro lado un segundo grupo con los campos que estaban vacíos cuando el envío del formulario **tuvo éxito** (la variable **`emptyInputsWhenSucceeded`**). De cualquiera de los campos en el primer grupo, (los que estaban vacíos en los envíos fallidos), se puede inferir con seguridad que no son obligatorios si también estuvieron vacíos en al menos un envío exitoso del formulario (envíos que se pueden obtener mediante el mensaje `#successfulSubmissions`). Entonces, para descartar estos casos, se substraen del primer grupo los campos del segundo. Notar que esta heurística no asegura que no se halle un falso positivo en el caso en que un campo que no es obligatorio nunca haya sido dejado en blanco. Esta información resultará de mucha utilidad al momento de aplicar el refactoring *Add Validation*.

11. Late Validation *validación tardía*

Este smell es esencialmente igual a *No Client Validation*, pero específico para el escenario en el que la validación ocurre luego de que el botón de "Enviar" del formulario es presionado, impidiendo el envío al servidor directamente. Esto suele significar que hay validación del lado del cliente, pero que puede mejorarse mediante validación por campo *in-line*, es decir, en el momento de abandonar cada campo individualmente para continuar con el siguiente. Este tipo de validación ha demostrado mejorar diferentes métricas como el éxito en el envío o los tiempos de llenado en algunos casos[13] (Wroblewski, 2008).

En realidad, ambos *finders* están modelados en una única clase, pero puede determinar los diferentes smells con la misma mecánica de detección.

12. Abandoned Form *formulario abandonado*

Cuando un formulario es demasiado complicado de llenar, es común que los usuarios lo abandonen. Si bien la complejidad del formulario no siempre es la causa, es una situación que vale la pena capturar. El usability smell *Abandoned Form* se detecta cuando la tasa de abandono supera un umbral, alertando al evaluador de un formulario que

[13] http://alistapart.com/article/inline-validation-in-web-forms (accedido 14 de junio de 2016)

probablemente sea demasiado complejo y desaliente a los usuarios. Durante evaluaciones de prueba, se obtuvieron buenos resultados con este umbral en un 0.4, pero éste es particularmente difícil de estandarizar, dado que los formularios web tienen diferentes propósitos: de un formulario de *login* se espera que tenga una tasa de éxito mucho más alta que de un formulario de contacto o de registro.

La detección de este smell se basa en contar proporciones de dos eventos diferentes: *Form Submission* y *Unfilled Form*. Estos dos eventos representan respectivamente los casos en los que un formulario se envía, sin importar el éxito del envío, y cuando se abandona. Si la tasa de abandono es superior al umbral especificado, se considera la presencia del smell *Abandoned Form*. La estrategia de agrupamiento es sobre un elemento DOM, que en este caso es el formulario afectado. Es importante recordar que los eventos *Unfilled Form* cuentan también con un valor de corte superior en el tiempo de permanencia, si se supera un umbral determinado el evento no se registra, dado que se considera que el usuario directamente abandonó la navegación para realizar alguna otra actividad.

Además de la información del porcentaje de rechazos, este usability smell también puede informar el tiempo promedio que los usuarios visualizaron el formulario antes de abandonarlo.

13. Scarce Search Results *resultados de búsqueda escasos*

Los formularios de búsqueda son una parte fundamental en la web, como lo indica Nielsen en su popular lista de 10 errores en el diseño web (top 10 mistakes in web design)[14]. El usability smell *Scarce Search Results* se detecta cuando un formulario falla en traer resultados la mayoría de las veces.

La manera de detectar este problema es mediante el análisis de eventos *Search*, que tienen la capacidad de informar si una búsqueda tuvo éxito o no, en términos de mostrar algún resultado. La proporción que está fijada por defecto es de 0.3. Cuando el usability smell se reporta (ver Figura 23), se muestra el formulario afectado junto con la proporción de búsquedas fallidas.

[14] https://www.nngroup.com/articles/top-10-mistakes-web-design

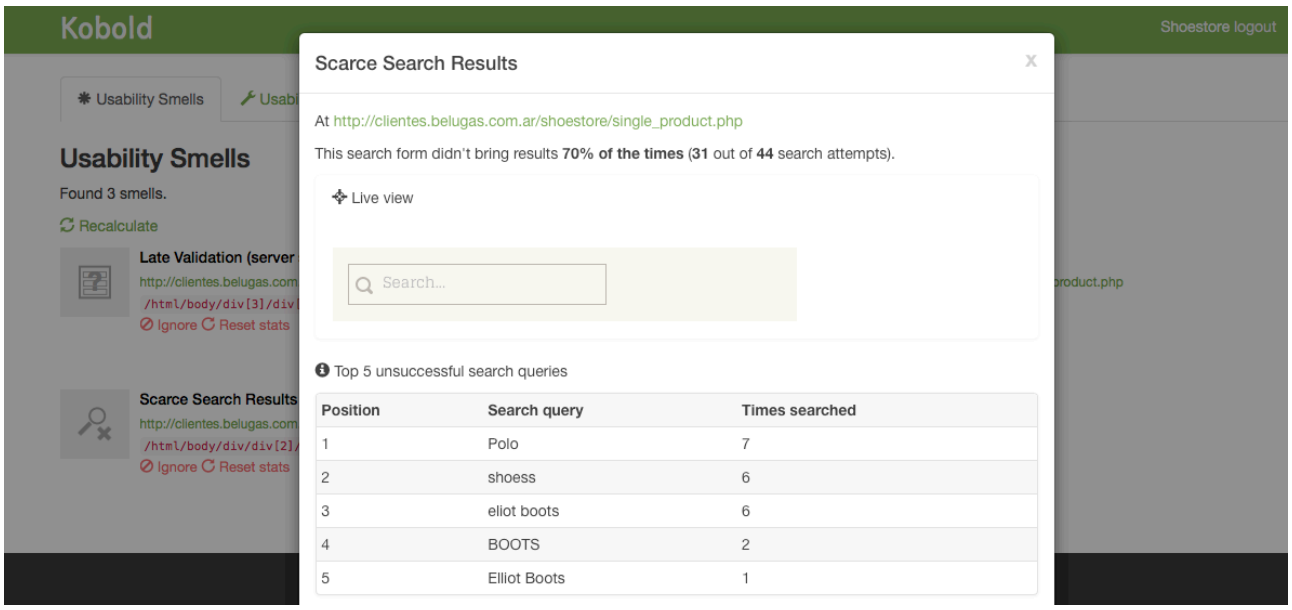

Figura 22. Reporte de un usability smell *Scarce Search Results*.

También informa sobre los términos de búsqueda fallidos más populares en un ranking, lo cual (ya fuera del ámbito de la usabilidad) puede incluso servir para ver las tendencias de búsquedas.

14. Useless Search Results *resultados de búsqueda inútiles*

Como una variante del smell previamente listado, *Scarce Search Results*, el usability smell *Useless Search Results* detecta lo que sucede luego de que un formulario de búsqueda sí trae resultados. Si los usuarios raramente hacen click en alguno de dichos resultados, entonces puede ser que éstos, si bien aparece, no sean los que el usuario estaba esperando.

La lógica de la detección para *Useless Search Results* es sencilla, dado que la tarea más compleja de la detección está a cargo de los eventos *Search*, que proveen mucha información útil. En este caso, basta con calcular una proporción de búsquedas en las que el usuario no sólo halló resultados, sino que hizo click en alguno de ellos (suponiendo un interés), y determinar si esta proporción supera un umbral determinado. Por defecto, y según las pruebas realizadas, el umbral prefijado es 0.3, pero puede depender del flujo de tráfico del sitio. Esta mecánica está basada en el trabajo de Dan et al., quienes afirman que una página de resultados de búsqueda es considerada exitosa y

el usuario hizo click en alguno de los resultados y no revisitó la misma página de búsqueda por al menos 30 segundos (Dan, Dmitriev and White, 2012).

Es importante tener en cuenta que, a veces, la página de resultados puede ser suficientemente informativa como para que ese click adicional en algún resultado de la búsqueda no sea necesario, y el smell puede ser ignorado en esos casos.

15. Wrong Default Value *valor por defecto incorrecto*

Completar un formulario puede ser tedioso, por lo cual tener la menor cantidad de preguntas posibles incrementa la probabilidad de que los usuarios de hecho lo completen. Tener valores por defecto razonables (según Wroblewski, "smart defaults" (Wroblewski, 2008)) para las preguntas reduce el tiempo de completado aún más.

El usability smell *Wrong Default Value* detecta si el valor más popular (es decir, el más frecuentemente seleccionado por los usuarios) en una lista de *radio buttons* o en un *select box* no coincide con el valor que se presenta por defecto. La mecánica de detección consiste en agrupar los eventos por el valor seleccionado para cada elemento DOM (es decir, lista de selección o conjunto de *radio buttons*). Si la proporción del valor mayoritario no es el valor por defecto, el smell se detecta.

Wrong Default Value está basado en el smell *Unnecessary activities in the main process* (actividades innecesarias en el proceso principal) de uno de los catálogos previos. En ocasiones, es normal que este smell sea ignorado si el desarrollador prefiere un valor por defecto por motivos particulares.

16. Unresponsive Element *elemento inmutable*

El usability smell *Unresponsive Element* se detecta cuando los usuarios hacen click reiteradamente sobre un elemento, pero éste no desencadena ninguna acción. Esto sucede cuando los elementos dan un incentivo para que los usuarios los presionen, debido a su apariencia. Elementos típicos donde se ha encontrado este smell incluyen fotos de productos, encabezados de página, etiquetas de *checkboxes* / *radio buttons* o textos subrayados.

La mecánica de detección consiste en contar los eventos *Click Attempt* que indican el comportamiento de hacer click sobre un elemento que no responde. Cuando la cantidad de eventos supera un umbral, el usability smell se captura. En este caso, como la cuenta de eventos es simple y no depende de una proporción, se agregó una ventana de tiempo para la medición. De esta forma se evita que la acumulación histórica de clicks sobre ciertos elementos produzca falsos positivos. El *finder* tiene una condición adicional para descartar elementos de gran tamaño, dado que rara vez los usuarios interpretan que un elemento de interfaz grande produce alguna acción, excepto en imágenes.

Como se mencionó previamente, la mayoría de los usability smells presentados en este catálogo son especializaciones de otros más genéricos presentados en un catálogo previo (Distante *et al.*, 2014). En la Tabla 1 se incluye una lista de todos los usability smells junto con los usability events que son utilizados en su detección.

Tabla 1. Usability Smells y su relación con los Usability Events.

Id	Usability Smell	Usability Event
1	**Undescriptive Element**	Tooltip Attempt
2	**Misleading Link**	Flash Navigation, Tooltip Attempt
3	**No Processing Page**	Long Request
4	**Free Input for Limited Values**	Text Input
5	**Unformatted Input**	Text Input
6	**Unmatched Size Input**	Text Input
7	**Forced Bulk Action**	Bulk Action
8	**Overlooked Content**	Flash Scrolling
9	**Distant Content**	Navigation Path
10	**No Client Validation**	Form Submission
11	**Late Validation**	Form Submission
12	**Abandoned Form**	Unfilled Form, Form Submission
13	**Scarce Search Results**	Search
14	**Useless Search Results**	Search
15	**Wrong Default Value**	Option Selection
16	**Unresponsive Element**	Click Attempt

Es importante recordar que los usability smells son indicadores o pistas de problemas, y que en algunos casos podrían no indicar un problema real de usabilidad, dependiendo de factores como el contexto. Por ejemplo, un *Wrong Default Value* podría sugerir que la selección más popular se convierta en el default, cuando el desarrollador en realidad necesita que sea otra (por ejemplo, al marcar deliberadamente la opción de recibir un newsletter).

Capítulo 6. Refactorings de Usabilidad

Fowler definió refactoring como un cambio que mejora la calidad del software, y también al proceso de aplicar esos cambios (Fowler, 1999). En su trabajo, catalogó diferentes transformaciones que producen mejoras en aspectos no funcionales del software como la legibilidad del código, la comprensibilidad, mantenibilidad y extensibilidad. Luego del libro de Fowler, muchos otros trabajos han extendido el catálogo de refactorings a otras áreas, y en algunos casos yendo más allá de la calidad interna, o imperceptible para los usuarios. Por ejemplo, existen refactorings para código HTML como "*Enable Caching*" (Harold, 2008), o refactorings sobre aplicaciones AJAX para migrar XML a JSON (Ying and Miller, 2013), ambos con el objeto de mejorar la performance. En este capítulo se describe la implementación de soluciones a *usability smells* en términos de refactoring.

En trabajos previos ligados a la presente propuesta, se han sugerido refactorings a nivel de modelo y código en aplicaciones web, con el objetivo de mejorar cualidades externas como usabilidad y accesibilidad (Garrido, Rossi and Distante, 2011; Garrido, Rossi, *et al.*, 2013). También se han catalogado varios refactorings específicamente diseñados para mejorar procesos de negocio en sitios de e-commerce (Distante *et al.*, 2014). Uno de estos refactorings es "*Keep the user up to date on the ongoing process*" (mantener al usuario informado sobre un proceso que está corriendo), apuntado a incrementar la confianza y satisfacción de los usuarios al momento de realizar un proceso, manteniéndolos al tanto de lo que sucede. Por ejemplo, este refactoring puede ser aplicado a un proceso de compra, mostrándoles qué es lo que han agregado al carrito mientras busca otros productos. En particular, en el trabajo sobre refactorings de accesibilidad (Garrido, Firmenich, *et al.*, 2013), se plantea el uso de *Client Side Web Refactorings* (CSWR), es decir, **refactorings web del lado del cliente**. Esta idea es fundamental para el trabajo aquí presentado, dado que es la única manera de modificar la interfaz (y en ciertos casos, incluso la navegación) de una aplicación web **sin alterar el código del servidor**, al cual no se tiene acceso.

El presente trabajo pretende lograr el máximo nivel de automatización posible en la mejora de usabilidad, y eso involucra también la generación y aplicación de refactorings de usabilidad web. Esto tiene 3 desafíos fundamentales a superar: la falta de especificidad, la multiplicidad.

En primer lugar, los refactorings provenientes de catálogos previos tienen a ser demasiados genéricos, y expresados en lenguaje natural, en ocasiones incluso con referencias a cómo presentar el contenido o qué tipo de lenguaje utilizar. Todo esto va en dirección opuesta al objetivo de plasmar los refactorings en código. Para resolver este problema, se analizó cada uno de los refactorings relevantes obtenidos de estos catálogos, y se los llevó a un lenguaje más concreto. Esto muchas veces resultó en la explosión de un refactoring a varios más específicos. Un claro ejemplo es el refactoring "Change Widget" (Grigera *et al.*, 2016), que propone reemplazar un elemento de interfaz por otro más apropiado para una acción. Esto por supuesto es imposible de implementar a tal nivel de genericidad, con lo cual se obtuvieron varios refactorings a partir de él, uno por cada widget a aplicar, como *Add Datepicker* o *Turn input into Radios*. De este proceso se obtuvo un nuevo catálogo de refactorings. Además, se agregaron algunos originales

que surgieron de la propia investigación, y la necesidad de resolver ciertos usability smells.

El segundo desafío es la multiplicidad de soluciones que puede haber para un mismo problema, más especialmente cómo decidir por una. No es el problema más grave, dado que siempre se pueden proponer muchas soluciones al usuario y que éste elija la mejor según su criterio, pero una herramienta capaz de decidir según el contexto y diagnóstico del problema sería de mucha más utilidad. Para lograr esto, en realidad se trabajó más sobre los usability smells que sobre los mismos refactorings. Esto se debe a que la herramienta principal para decidir entre varios refactorings es el detalle del diagnóstico del smell.

Tomemos por ejemplo el smell *Free Input for Limited Values*. Este smell determina que un campo de texto libre permite ingresar cualquier valor, cuando en realidad los valores posibles que se pueden ingresar corresponden a un rango cerrado, como países u ocupaciones. Existen 3 refactorings para solucionar este smell: *Add Autocomplete*, *Turn Input into Select*, y *Turn Input into Radios*.

El primero, *Add Autocomplete,* simplemente provee asistencia a medida que se ingresa el texto, con un menú de opciones predefinidas que aparece a medida que se encuentran coincidencias en una base de datos.

El segundo, *Turn Input into Select*, reemplaza directamente el campo de texto por una serie de opciones predefinidas, incluyendo la posibilidad de ingresar una opción que no esté en el listado. Si el diagnóstico simplemente indica que un campo de texto está afectado por el smell sin mayor detalle, sería muy difícil determinar qué solución es mejor, sin embargo, conociendo el número y la proporción de los valores frecuentemente ingresados, se puede optar con más seguridad por uno de ellos.

En este caso, cuando los valores frecuentes son relativamente pocos (8 o menos es un buen criterio), la opción de convertir el campo en un listado de opciones con radio buttons es generalmente más cómoda para los usuarios, dado que no hay que escribir texto, y además las opciones están visibles. Cuando los valores frecuentes tienden a ser más (entre 9 y 20), la opción de *Turn Input into Select* puede ser más cómoda. Si bien la visibilidad es menor que en el caso anterior, se agrega la posibilidad de ingresar las primeras teclas de un valor para hallarlo dentro de la lista. En este caso hay que

considerar la salvedad de que no pueden incluirse valores que no estén en el listado, con lo cual sólo se puede aplicar si todos los valores ingresados están en el conjunto de valores frecuentes.

Por último, cuando los valores frecuentes son muchos (más de 20 aproximadamente), tiene menos sentido establecer un listado fijo de opciones, ya que es más simple directamente ingresar lo que se quiere que buscarlo en una lista de gran tamaño. Al agregar un autocompletado con el refactoring *Add Autocomplete* da la doble ventaja de ayudar al usuario a escribir menos texto, y de darle la confianza de estar ingresando un valor contemplado.

6.1. Implementación

Los refactorings de usabilidad se implementan en JavaScript, dado que necesariamente deben ser ejecutados del lado del cliente íntegramente (son en definitiva CSWR – Client Side Web Refactorings). El código se genera automáticamente a partir de la información provista por el usability smell que se intenta resolver.

Al momento de la escritura de este trabajo, la implementación de Kobold cuenta con 8 refactorings automatizados, que pueden generarse con la información de los usability smells, aunque en algunos pocos casos requiere de ayuda por parte del usuario. Estos últimos casos son denominados refactorings semi-automatizado, y están diseñados para que cualquier usuario puede completar la información faltante, sin importar su experiencia en usabilidad (y sólo un conocimiento básico de HTML, que se asume dado que de todas formas los usuarios de Kobold son *a priori* desarrolladores).

La generación de código es en sí una tarea algo engorrosa, debido al cuidado que requiere, por ejemplo, al momento de escapar caracteres. Por esto existen en el framework algunas clases que ayudan a esta tarea. En la Figura 23 se observa un modelo parcial de las clases que representan los refactorings web.

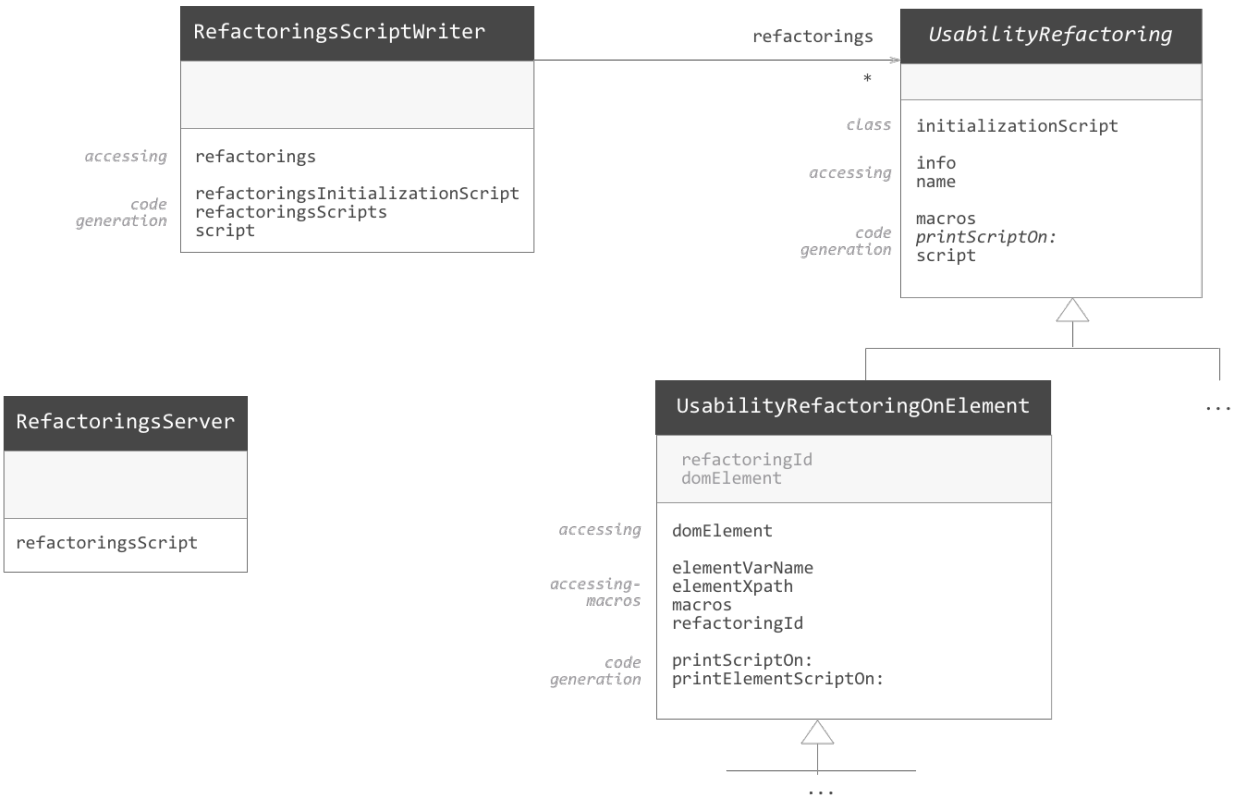

Figura 23. Modelo parcial de Usability Refactorings.

La clase que inicia el proceso de generación de refactorings es `RefactoringsServer`. Es en realidad un servicio RESTful que es invocado desde el componente del lado del cliente cada vez que se carga. El llamado que recibe el `RefactoringsServer` es un pedido de refactorings de parte de una cuenta de usuario (a través de un token, el mismo que se utiliza para identificar la cuenta de usuario al momento de capturar usability events). Una vez obtenida la cuenta de usuario correspondiente, se piden los usability refactorings que se hayan instanciado para dicha cuenta, y esta información es enviada a un generador de código JavaScript modelado por la clase `RefactoringsScriptWriter`.

Cada instancia de la clase `RefactoringsScriptWriter` es creada con un conjunto de refactorings, y ésta se encarga de generar el código JavaScript completo para todos ellos mediante el método `#script`, que a su vez llama a otros dos métodos:

- `#refactoringsInitializationScript` se ocupa de cargar el código único para cada familia de refactorings. Por ejemplo, si se quieren aplicar dos instancias del refactoring *Add Date Picker* (que agrega un selector de fechas de tipo calendario),

se debe importar la librería JavaScript que lo permite, pero sólo una vez. Para esto, cada clase de refactoring implementa `#initializationScript`, que es con un mensaje de clase. Este método es llamado dentro del método `#refactoringsInitializationScript` una vez por cada clase de refactorings involucrada. En algunas clases, si el refactoring es muy simple, el método `#initializationScript` queda sin comportamiento.

- `#refactoringsScripts` genera el código particular requerido para cada instancia del refactoring. Siguiendo con el ejemplo del *Add Date Picker*, aquí es donde se genera el calendario para cada campo de texto que se quiere refactorizar.

Dentro de cada refactoring, existe una mecánica de generación de código JavaScript implementada dentro del método `#script`. Este método es abstracto en la clase raíz de la jerarquía `UsabilityRefactoring`, pero contiene una implementación básica de gran utilidad para los refactorings que se aplican sobre un elemento DOM, que están implementados como subclases de `UsabilityRefactoringOnElement`. Esta clase intermedia implementa la generación del script agregando la declaración en JavaScript de una variable que contiene el elemento (o la familia de elementos) afectado por el usability smell correspondiente. El código generado utiliza el *xpath* del elemento para recuperarlo del modelo DOM, y lo convierte en un objeto jQuery para facilitar su manipulación posterior. A continuación, se muestra un ejemplo de este código generado:

```
var element2db3bee8e7100d00a4bab16a0fda3de9 =
   $(xpathInstance.getElementByXpath(
      '/html/body/div[3]/div[2]/div/div/div[2]/form'
   )
);
```

El código es en esencia muy simple, pero tiene varias partes para destacar. Lo primero que llama la atención es el extenso nombre de la variable donde se guarda el elemento DOM: `element2db3bee8e7100d00a4bab16a0fda3de9`. La longitud de dicho nombre se debe a que incluye un identificador único (la cadena de caracteres posterior a la palabra “element”) generado para ese elemento, para evitar completamente cualquier posibilidad de potenciales colisiones con otras variables. La parte derecha de la asignación se divide

en dos: el código para obtener el elemento a partir de su *xpath* mediante la función `getElementByXpath` del objeto `xpathInstance`, que reúne toda la funcionalidad referida a *xpaths*. Lo último que se hace, luego que se recuperó el elemento, es convertirlo en un objeto jQuery mediante la función estándar abreviada que lleva como nombre el sigo "peso" - `$()`.

Cada subclase de `UsabilityRefactoringOnElement` puede hacer uso de esta variable, pudiéndose generar sólo el código para manipular el elemento (o los elementos) DOM. Sólo un refactoring de los implementados hasta el momento no hace uso de esta característica, modelado en la clase `AddProcessingPage`, que es subclase directa de `UsabilityRefactoring`.

En general, la generación de código está técnicamente basada en el uso de dos herramientas para la manipulación de strings: *streams* y *macros*. Los *streams* son objetos que facilitan la concatenación de *strings*, facilitando la generación de código enormemente. Estos objetos funcionan como canales que consumen strings uno a uno, y luego se les puede pedir el resultado completo. Esta mecánica produce un código mucho más simple que la constante concatenación de strings en múltiples asignaciones incrementales. Los *macros* son cadenas especiales que se pueden introducir en un String para luego ser reemplazadas (por ejemplo, la función `printf` del lenguaje C). Es un concepto simple, pero de gran utilidad para evitar constantes interrupciones en la escritura del código a generar (en este caso JavaScript), particularmente cuando se quiere incorporar muchas veces un mismo texto generado dinámicamente. En el caso de las subclases de `UsabilityRefactoringOnElement`, es de gran utilidad para todas las referencias a la variable del elemento DOM afectado.

6.2. Catálogo de Usability Refactorings

En esta sección se muestra el catálogo de refactorings implementados al momento de escritura de este trabajo. Para cada uno se muestran detalles de la generación de código, y se indica su nivel de automatización.

I. Add Processing Page

El usability refactoring *Add Processing Page* se encarga de agregar una página de proceso a una acción que normalmente lleva mucho tiempo. El usability smell que intenta resolver es *No Processing Page*, que indica un proceso largo que no anuncia al usuario que se está llevando a cabo.

La implementación de *Add Processing Page* sirve para lo casos en que el smell *No Processing Page* afecta a un formulario. Este refactoring se realiza en varios pasos:

1. El botón de *submit* del formulario en cuestión se oculta, y se reemplaza por un nuevo botón, generado con las mismas características para imitarlo. Para lograr este efecto, se obtiene el estilo CSS y contenido del botón original y se aplican al botón nuevo, que se genera en realidad clonando el original con la función `clone()` de jQuery.
2. Se genera un *overlay*, que es un panel translúcido para crear el efecto de oscurecimiento de la pantalla, ver Figura 24.
3. Se asignan dos acciones al nuevo botón de envío o *submit*. En primer lugar, muestra el *overlay*. Inmediatamente después, envía el formulario programáticamente.

Un detalle técnico interesante sobre este proceso está en cómo se logra que el *overlay* se llegue a mostrar antes del envío del formulario. Para lograr este efecto se utiliza la función asincrónica de JavaScript `setTimeout()` con valor de 1 milisegundo. Este es una *workaround* estándar de JavaScript para asegurar que las dos acciones se ejecuten en secuencia, es decir, que siempre se muestre el *overlay* antes del evento *submit*, que tiene una tendencia a capturar el procesamiento e impedir que se ejecuten otras acciones. Este mismo motivo es el que impide implementar el refactoring utilizando el botón de *submit* original, ya que es prácticamente imposible ejecutar código una vez que la secuencia de envío se inició.

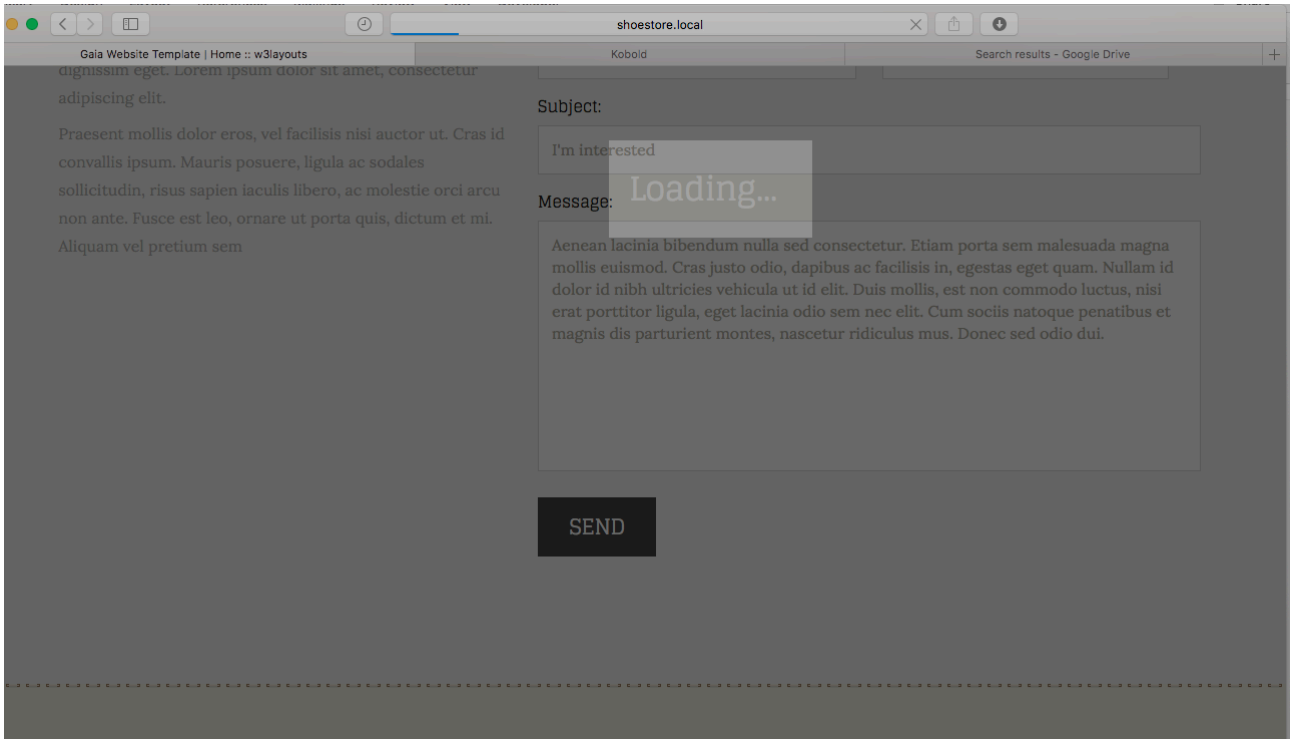

Figura 24. Usability Refactoring Add Processing Page funcionando. En la captura se puede ver el *overlay* generado automáticamente, en el momento en que se está enviando el formulario.

El refactoring *Add Processing Page* es completamente automático y no necesita que el desarrollador complete con ninguna información, aunque se podrían agregar opciones de personalización, por ejemplo, para el estilo del *overlay* o del clon del botón original.

Es importante notar que este refactoring es el único que no está implementado como subclase de `UsabilityRefactoringOnElement`, dado que no afecta un elemento en particular sino una acción de navegación, o más específicamente, un proceso. Otra característica particular es que no requiere importar librerías, más que jQuery.

II. Add Autocomplete

El refactoring *Add Autocomplete* se ocupa de generar un autocompletado sobre un campo de texto, y puede utilizarse para solucionar dos usability smells: *Free Input for Limited Values* y *Scarce Search Results*.

En el caso del *Free Input for Limited Values*, que recordemos, detecta que un campo de texto libre permite en realidad un conjunto limitado de valores, el refactoring *Add Autocomplete* se puede elegir cuando el número de opciones es muy amplio, y no se

quiere restringir estrictamente, es decir, se quiere permitir que se ingresen valores fuera de los más populares.

En el caso de aplicarse *Add Autocomplete* para resolver el usability smell *Scarce Search Results*, la idea es ayudar al usuario a reafirmar que lo que está buscando realmente existe.

La implementación de este refactoring tiene dos partes: por un lado, se importa una librería JavaScript capaz de generar un *autocomplete* (para esta implementación se utilizó la librería *Awesomeplete*[15] por su simpleza y por no requerir ninguna dependencia), y luego se genera un código de inicialización para el campo de texto afectado. Gracias a la implementación provista por la superclase `UsabilityRefactoringOnElement`, la generación de JavaScript escrita en la clase que implementa *Add Autocomplete* es muy simple, como se puede observar en el siguiente código:

```
printElementScriptOn: aStream
   aStream nextPutAll:
     'new Awesomplete({elementVarName}[0], \{list: [{valuesList}]});'
```

El código muestra la gran ayuda que provee el uso de *macros*. En este caso se observan dos: la primera es `elementVarName`, que está definida en la superclase como se explicó previamente. La segunda *macro* es `valuesList`, que se reemplaza luego con la lista de valores implementada en el método del mismo nombre:

```
valuesList
    ^ String streamContents:
        [:stream | self values
            do: [:value | stream nextPutAll: value printString ]
            separatedBy: [ stream nextPutAll: ',']
        ]
```

este código genera un string con una lista de valores para crear el array JavaScript. El método `values` es el que realmente retorna la colección de valores a utilizar en el *autocomplete*. En el caso del smell *Free Input…* obtiene esos valores automáticamente de los conjuntos de valores populares, pero el usuario puede completar o corregir antes de aplicarlos (ver una captura del momento de su aplicación en la Figura 25).

[15] Awesomeplete: http://leaverou.github.io/awesomplete/

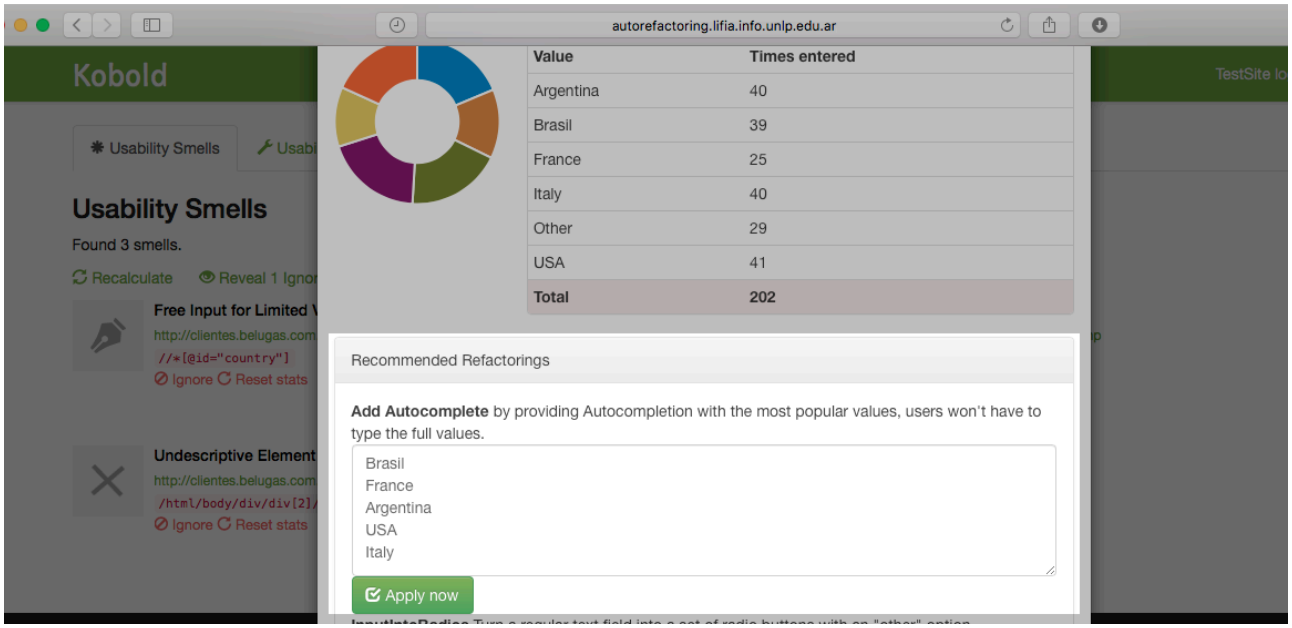

Figura 25. Captura de Kobold ofreciente aplicar el usability refactoring *Add Autocomplete* para resolver un usability smell *Free Input for Limited Values*. Se puede ver que genera la lista de posibles valores de los valores populares del smell, pero permite al usuario editarla.

Este refactoring puede considerarse como automático o semi-automático dado que el usuario puede o no alterar los parámetros antes de la generación.

III. Add Date Picker

Este refactoring es una de las variantes específicas del refactoring *Change Widget* originalmente reportado en catálogos previos (Grigera *et al.*, 2016). Sirve como una de las variantes para resolver el usability smell *Unformatted Input* para el caso del ingreso de fechas. Es importante destacar que es sólo una variante de resolución, porque el smell mencionado puede ser tener diferentes soluciones dependiendo del tipo de fecha que se solicita. Por ejemplo, el refactoring *Add Date Picker* puede ser útil para ingresar fechas de reserva de un hotel, donde es importante ver el calendario completo que muestra los días de la semana, pero podría ser una carga extra para cuando hay que ingresar una fecha que los usuarios conocen de memoria, como la fecha de nacimiento. En ese caso un refactoring como *Format Input* puede ser más adecuado, dado que permite a los usuarios ingresar la fecha directamente desde el teclado, pero agrega la seguridad de estar ingresando mes día y año en el orden adecuado.

La implementación de *Add Date Picker* es relativamente simple, ya casi todo el código necesario se encuentra en una librería (en la implementación de Kobold se optó

por la librería Datepickr[16]), y para cada instancia del refactoring sólo hace falta inicializar el campo afectado:

```
printElementScriptOn: aStream
    aStream nextPutAll:
     'datepickr({elementVarName}[0], \{ dateFormat: "d/m/Y"})'
```

Como puede verse en el código, sólo se llama a la función datepickr que genera el widget necesario, especificando el formato de fecha "d/m/Y". Este refactoring se considera automático, ya que no hace falta proporcionar información adicional. Podría, sin embargo, agregarse opciones de personalización para alterar el formato o cambiar el estilo del calendario.

IV. Add Form Validation

Existen dos usability smells similares que indican que un formulario no muestra los mensajes de validación a tiempo, ya sea luego de que el usuario ya presionó el botón de *submit*, detectado por el smell *Late Validation*, o bien luego de que el envío se hizo al servidor, representado en el smell *No Client Validation*.

El usability refactoring *Add Form Validation* se ocupa de agregar validación *inline*, es decir lo más inmediatamente posible, luego de que el usuario retira el cursor de cada campo de texto (acción que se puede capturar con el evento JavaScript *blur*). Por el momento, la validación que el refactoring es capaz de realizar es sobre campos requeridos, aquellos que no deberían dejarse en blanco. Los usability smells que capturan la validación tardía tienen forma de inferir cuáles de los campos son obligatorios, como se explicó en el Capítulo 5.

La generación de código no requiere de librerías externas, sólo se genera una porción de código JavaScript por cada formulario afectado, en cada instancia del refactoring, modelado en la clase del mismo nombre (`AddFormValidation`). El código que se genera realiza varias acciones:

1. Para cada campo que se considera obligatorio, se agregan acciones al momento de *blur* (retiro del cursor). Se observa si el campo está vacío y en

[16] Datepickr – pick your date not your nose https://github.com/joshsalverda/datepickr

ese caso se destaca con color rojo el borde del campo, lo que es un código prácticamente universal para indicar que hay un problema con el campo, y si bien no incluye un mensaje tiene la ventaja de no estar atado a un idioma.

2. Para cada campo obligatorio, se agrega la marca "(*)" (asterisco), que también es interpretada por los usuarios como un campo requerido.
3. Se intercepta el evento *submit* del formulario afectado, para poder revisar que los campos considerados obligatorios no estén vacíos, más allá de que se haya informado al usuario en el momento de abandonar cada uno de ellos. Si alguno estos campos no se completó, se impide el envío del formulario.

Este usability refactoring es considerado automático, porque el usuario desarrollador no necesita agregar información, y todo está inferido automáticamente.

V. Date Input Into Selects

Como se mencionó en la descripción del refactoring *Add Date Picker*, hay múltiples maneras de resolver el usability smell *Unformatted Input* para el caso particular del ingreso de fechas. Una de ellas consiste en transformar la entrada de campo libre en un conjunto de 3 listas de selección: una para el día del mes, otra para el mes, y otra para el año. Este tipo de selección de fechas es efectiva para fechas que el usuario puede recordar de memoria, pero tiene la ventaja de no requerir validación, ya que las listas no permiten introducir fechas inválidas.

La generación de código contiene una pequeña librería jQuery para facilitar la instanciación en el caso de necesitar refactoring en múltiples campos. Esto produce que el código para inicializar un campo en particular sea muy simple:

```
printElementScriptOn: aStream
    aStream nextPutAll: '{elementVarName}.dateDropdowns();'
```

Es importante recordar que la expresión `elementVarName` es en realidad una *macro* que antes de terminar la generación se reemplaza con el nombre real de la variable que liga al elemento (en este caso, el campo de texto).

El refactoring *Date Input Into Selects* es automático, pero podrían agregarse algunas opciones durante su aplicación, por ejemplo, para restringir los años que se muestran o cambiar el idioma de los meses o incluso el orden.

VI. Turn Input Into Radios

El usability smell *Free Input for Limited Values* indica que un campo de texto libre está siendo utilizado para solicitar datos que corresponden a un conjunto limitado de valores. Cuando este conjunto de valores es muy pequeño, y los valores populares muy recurrentes, y además el grupo de *outliers*, es decir valores extraños a estos grupos, es muy pequeño o está directamente vacío, se puede optar por el usability smell *Turn Input Into Radios* (ver un ejemplo en la Figura 26).

Figura 26. Ejemplo de refactoring *Turn Input Into Radios*. A la izquierda se puede ver el campo de texto para ingresar un país ("Country"), y a la derecha el refactoring aplicado con las opciones populares, pero agregando la posibilidad de ingresar otro valor.

Como su nombre lo indica (convertir campo de texto en *radio buttons*), el refactoring reemplaza el campo de texto original afectado por el usability smell, por una lista de *radio buttons* con las opciones populares. Cuando las opciones son pocas, esta manera de presentar la entrada es más cómoda para el usuario, dado que no hace falta ninguna acción para visualizarlas: en las listas de selección sólo se ven al abrirse el menú desplegable, y en el autocompletado, no se ven las opciones hasta que no se ingresan algunos caracteres, y ni siquiera en ese caso se ven todas. Cuando hay muchos valores, en cambio, la lista de *radio buttons* es inapropiada, ocupa demasiado espacio de pantalla y complica a los usuarios que tienen que leer cuidadosamente una lista larga para encontrar la opción indicada.

La implementación del refactoring, si bien es algo compleja, no depende de una librería externa (además de jQuery). El cambio consiste en ocultar el campo de texto

afectado por el *Free Input…* y en su lugar generar el código HTML necesario para la lista, con las opciones populares extraidas del usability smell. Se agrega también una opción "other/otro" para que el usuario siempre pueda ingresar algún valor que no está entre las opciones, sin alterar la funcionalidad original. Los *radio buttons* están programados para que cuando se seleccione una opción, se carge el contenido de la etiqueta en el campo de texto original (ahora oculto), y de esa manera se asegura que todo funciona de la misma manera, aunque el usuario lo perciba diferente.

Este refactoring, al igual que *Add Autocomplete* puede verse como automático o semi-automático, dado que también se les da a los usuarios la posibilidad de editar el listado de valores populares. Es también un caso específico del refactoring *Change Widget* (Grigera *et al.*, 2016).

VII. Provide Default Option

El refactoring *Provide Default Option* es muy sencillo, tanto en idea como en implementación. Simplemente consiste en cambiar la selección por defecto de un componente de tipo *select box* o bien un listado de *radio buttons* afectados por el usability smell *Wrong Default Option*.

Implementativamente, lo único que hace falta hacer es cambiar la selección por el valor por defecto obtenido del usability smell. El código que se genera, en el caso de refactorizar un *select box* es el siguiente:

```
printElementScriptOn: aStream

    aStream nextPutAll:
       '{elementVarName}.prop("selectedIndex", {defaultOptionIndex});'
```

Una vez más, se nota el uso de las *macros* para acceder al elemento DOM a refactorizar (**`elementVarName`**) y para obtener el valor por defecto (**`defaultOptionIndex`**). No se requiere ninguna librería, más allá de jQuery, que se necesita de todas maneras para la funcionalidad básica de todo el componente del lado de cliente de la herramienta.

Este refactoring es automático, dado que el usuario no necesita completar con ninguna información adicional, y todo está provisto por el usability smell.

VIII. Turn Attribute Into Link

Cuando un elemento DOM está afectado por el usability smell *Unresponsive Element*, significa que los usuarios frecuentemente intentan realizar alguna acción al hacer click en él, pero nada sucede.

El usability refactoring *Turn Attribute Into Link* ofrece al desarrollador la oportunidad de convertir el elemento en un link. Es también muy simple en implementación, y tampoco requiere de ninguna librería adicional, pero necesita de la ayuda del desarrollador para indicar cuál es el destino del link. Al presentar la sugerencia de refactoring, la herramienta Kobold muestra un pequeño formulario donde se puede indicar la URL (ver Figura 27).

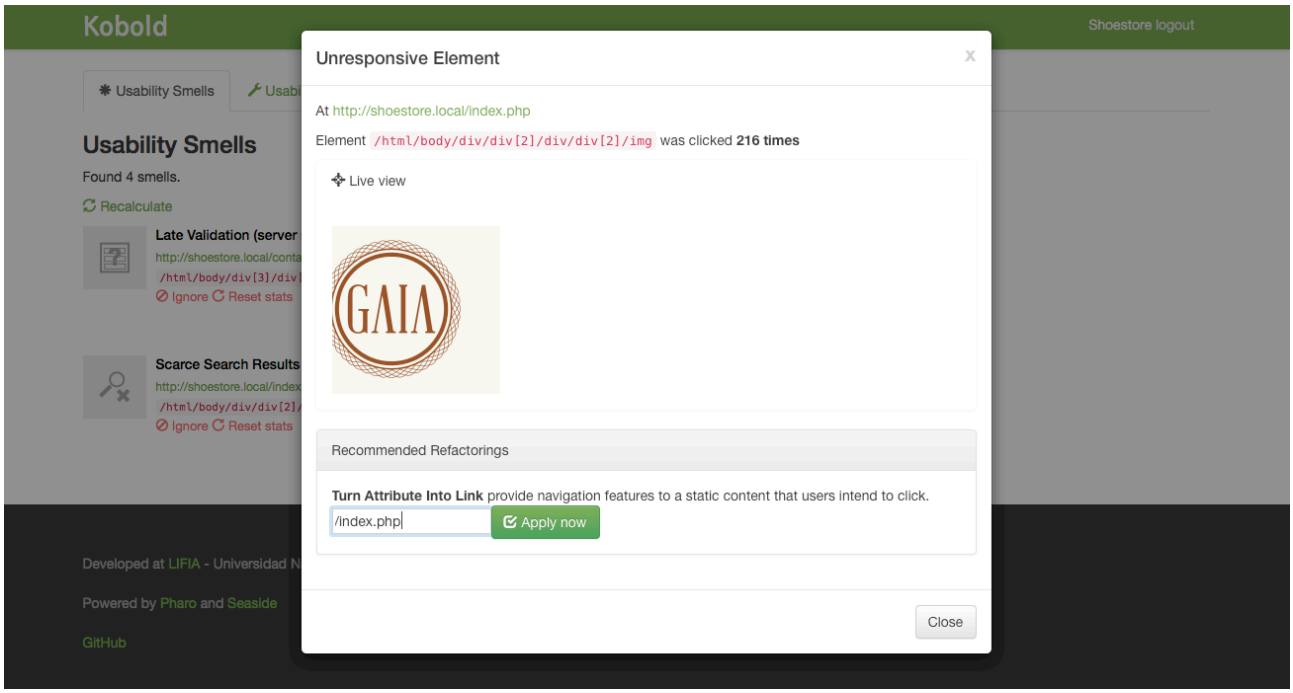

Figura 27. Captura mostrando el reporte de un usability smell *Unresponsive Element* y la sugerencia de aplicación del refactoring *Turn Attribute Into Link*. Se puede ver el xpath del elemento afectado (un encabezado), una vista del mismo, y el formulario para agregar el destino del nuevo link.

Este es un refactoring semi-automático, dado que el usuario completa con información necesariamente, ya que no es posible que el usability smell infiera cuál debería ser el destino del link.

IX. Resize Input

Como se explicó anteriormente en la descripción del usability smell *Unmatched Input Size*, el ancho de los campos de texto puede ayudar a los usuarios a tener una mejor idea del

tipo de contenido que se espera. Contrariamente, cuando la longitud del campo de texto es muy diferente de los textos que se suelen ingresar, esto puede confundirlos.

Al detectar un *Unmatched Input Size*, se indica que la longitud promedio de los textos ingresados es, o bien mucho mayor o mucho menor que el tamaño del campo. Esto es un inconveniente más grave en el último caso, dado que parte de lo ingresado queda oculto, dificultando al usuario la posibilidad de revisar lo que completó.

El refactoring *Resize Input* permite alterar la longitud de un campo de texto para que se ajuste mejor a los textos típicamente ingresados. Es totalmente automático, dado que para configurar la nueva longitud obtiene el promedio de longitud de los textos ingresados (del smell en cuestión). A esta longitud promedio se suma el desvío estándar de las longitudes ingresadas. Esto hace que el nuevo tamaño esté mejor preparado para una mayor cantidad de casos. Por ejemplo, si el desvío estándar es muy pequeño, esto significa que la longitud de los textos ingresados es más uniforme, con lo cual, sumar este agregado a la nueva dimensión no aleja mucho el tamaño de la media. Si, en cambio el desvío estándar es mayor, significa que los textos pueden ser considerablemente más largos, y la nueva longitud del campo también queda dimensionada acorde.

X. Rename Element

Cuando un elemento interactivo, como un botón o un link, está afectado por el smell *Undescriptive Element*, esto quiere decir que probablemente los usuarios tengan dificultades comprendiendo cuál es su propósito. Una forma sencilla de resolver este problema podría ser alterando su contenido para explicar más claramente lo que sucede.

El refactoring *Rename Element* permite exactamente eso, dejando al desarrollador elegir el nuevo contenido del link o botón, opcionalmente permitiéndole ingresar código HTML (de gran utilidad para agregar o alterar un ícono, por ejemplo). Esta posibilidad clasifica al refactoring como semi-automático.

XI. Link to top

Este refactoring agrega un botón para retornar al tope de la página mediante una acción de scroll (ver Figura 28). Estos botones son populares en los sitios que tienen contenido de extensión vertical, como el diseño de *parallax* muy utilizado en los últimos años.

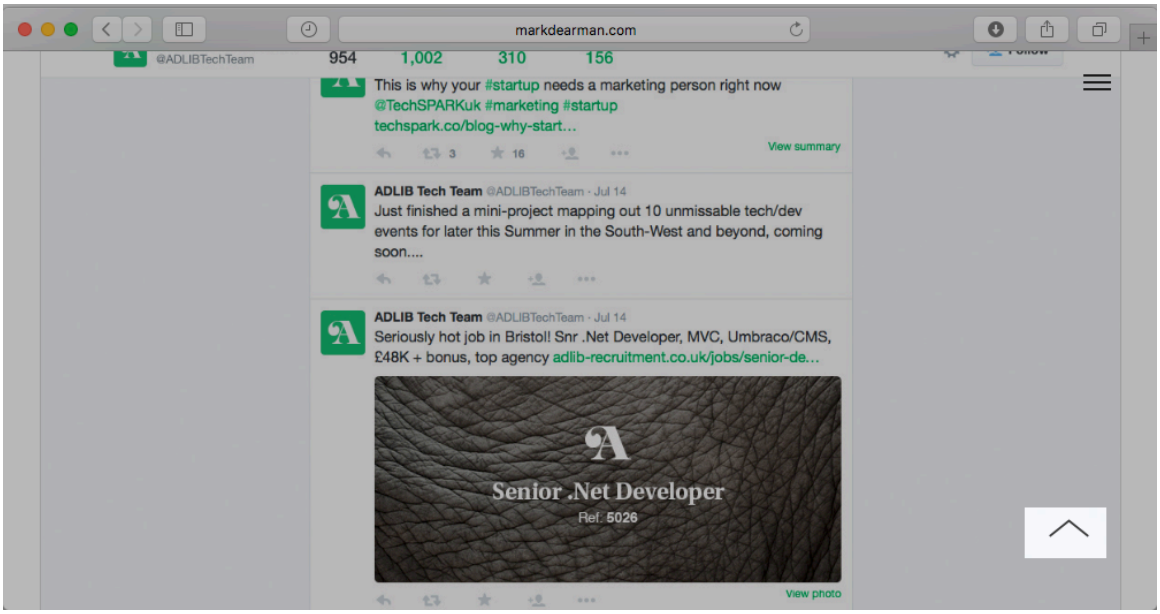

Figura 28. Ejemplo de botón para regresar al tope de la página.

El usability smell que este refactoring resuelve es *Overlooked Content*, que indica múltiples acciones rápidas de scroll. Esta solución sólo es útil para un caso particular del smell, cuando las acciones de scroll son hacia el tope de la página.

El refactoring *Link to Top* es completamente automático, y no requiere de ningún dato adicional por parte del desarrollador dado que su propósito es simple.

XII. Add Tooltip

Este refactoring puede servir para resolver algunos casos del usability smell *Undescriptive Element*. Dado que el diagnóstico de este smell ya está basado en el uso de tooltips, no siempre tiene sentido aplicar este refactoring ya que un tooltip puede ya existir. Sin embargo, puede haber dos motivos válidos para instanciar *Add Tooltip*: cuando no se desea cambiar el contenido del elemento afectado por algún motivo, y cuando el elemento afectado es un ícono por motivos de diseño o de espacio. En este caso, si bien es posible reemplazar el ícono mediante el refactoring *Rename Element*, el tooltip da una ayuda, y al ser instantáneo no hace perder mucho tiempo al usuario.

Este refactoring necesita importar la librería Tipr[17], que es un plugin de jQuery de muy bajo overhead. El desarrollador debe indicar el texto que contiene el tooltip, con lo cual es un refactoring semi-automático.

[17] Tipr - http://www.tipue.com/tipr/

XIII. Add Link

El refactoring *Add Link* sirve para agregar un atajo de navegación. El usability smell que intenta reparar es *Distant Content*, que señala una navegación de más de dos nodos muy frecuente entre los usuarios.

Para implementar este refactoring, se agrega el link al destino final al lado del link original, es decir, el primer link donde se origina el camino de navegación. Es un refactoring semi-automático, dado que el desarrollador debe proveer el contenido del link.

XIV. Distribute Menu

El usability smell *Forced Bulk Action* indica que los usuarios deben seguir un complicado flujo de pasos para aplicar una simple acción sobre un ítem en una lista. Este flujo es óptimo para aplicar la acción a muchos elementos a la vez, pero para uno solo resulta complejo. Una manera de resolver este usability smell es agregando una o varias acciones a cada elemento de la lista, cerca del mismo, para que sea más sencillo aplicar la acción si lo que se quiere es afectar a un solo elemento.

Si bien existen maneras de automatizar este refactoring, el diseño de este tipo de interacción es extremadamente variado en la forma de aplicar las acciones: en algunos casos hay un botón por acción, en otros la acción se aplica al seleccionarla de una lista, o combinaciones de ambos. Gmail, por ejemplo, sólo muestra los botones una vez que se seleccionó un elemento. Esta variedad sumada a las múltiples maneras no estandarizadas de implementar estos componentes, hace muy difícil la detección de la acción, y por lo tanto su reproducción en un eventual refactoring automatizado del lado del cliente. Por este motivo, el refactoring es sólo sugerido.

XV. Split Page

En ocasiones, puede detectarse que una página está muy cargada de contenido. El smell *Overlooked Content* puede indicar esto, dado que los usuarios están ignorando parte del contenido. Otro ejemplo de carga excesiva de contenido es lo que podría detectar *Abandoned Form*, cuando un formulario normalmente no termina de completarse.

En ambos casos, el refactoring *Split Page* podría servir para dividir un contenido demasiado extenso en múltiples páginas separadas, vinculadas entre sí de alguna

manera. En el caso del contenido para lectura (sin interacción considerable), la división puede hacer simplemente mediante links, creando una estructura de navegación. Cuando se trata de un formulario demasiado largo, la división de contenido puede hacerse llevando el formulario único a una serie de pasos con formularios más cortos.

El caso de separar contenido puro, el refactoring es difícil de automatizar, dado que tendría que interpretarse de alguna manera el contenido en sí para crear diferentes segmentos significativos. En el caso de separar formularios en sub-formularios, automatizar la tarea con un refactoring del lado del cliente puede ser complejo dado que se requiere usualmente trabajar con el código en el servidor. Por estos motivos, el refactoring *Split Page* es sólo sugerido.

6.3. Relaciones entre Usability Events, Usability Smells, y Usability Refactorings

En este capítulo se presentó un catálogo de usability refactorings ya implementados, es decir que ya han probado que pueden ser aplicados ya sea automáticamente a partir de la información capturada por el usability smell que resuelve, o con información agregada por el usuario, es decir, semi-automáticamente. En la Figura 29 se puede observar la relación que hay entre todos los usability events, usability smells y usability refactorings.

Figura 29. Lista esquemática con las relaciones entre todos los usability events, usability smells y usability refactorings.

A la izquierda se muestran los usability events, indicando con flechas para cuáles de los usability smells (centro) se utilizan en la detección. A la derecha, los usability refactorings en el catálogo de esta sección, relacionados con los usability smells que resuelven. Los refactorings XIV y XV están en un tono más claro para diferenciarlos de los que tienen algún nivel de automatización.

Notar que tanto las relaciones entre eventos y smells, como las que hay entre smells y refactorings, son muchos a muchos.

Capítulo 7. Validación

Para validar la metodología y su implementación a través de la herramienta Kobold, se realizaron varios experimentos para las diferentes etapas. En este capítulo se detallan las validaciones realizadas para comprobar la utilidad de los usability refactorings previo a la implementación de Kobold, y también se describen validaciones para la captura de eventos, la detección de usability smells y finalmente la aplicación automatizada de usability refactorings. Finalmente, se muestra una validación para comprobar el funcionamiento del algoritmo Scoring Map utilizado para agrupar eventos en muchos de los buscadores de usability smells descritos en el Capítulo 5.

7.1. Validación de Refactorings Manuales

La factibilidad de la aplicación de refactorings de usabilidad y su efecto, fueron estudiados previamente a la implementación de Kobold. En un trabajo de análisis estadístico con

voluntarios, se buscó validar mediante diferentes hipótesis que los refactorings de usabilidad ofrecen mejoras en sus tres aspectos: efectividad, eficiencia y satisfacción (Grigera *et al.*, 2016). En esta sección se detalla una evaluación empírica sobre cómo perciben los usuarios el impacto de los refactorings de usabilidad en aplicaciones web.

7.1.1. Definición y preparación del experimento

El experimento se condujo en 2 replicaciones, una en Argentina (en adelante AR) y otra en España (en adelante, SP). En el experimento se comparó la usabilidad en dos aplicaciones web, **antes** y **después** de aplicar refactorings. Como aplicaciones se utilizaron un sitio de e-commerce y uno de subastas. En ambos casos, se instalaron productos "enlatados" en un ambiente controlado (Zencart[18] y WeBid[19] respectivamente).

Se reclutaron voluntarios tanto en la replicación AR como en SP, para participar en rol de usuarios finales de las aplicaciones. La elección de diferentes localizaciones permitió comparar dos perfiles de usuario diferentes: desarrolladores de software (AR) y usuarios finales regulares, es decir, sin un entendimiento del funcionamiento de una aplicación por sobre el estándar (SP).

Para medir el efecto que cada refactoring tuvo en la usabilidad, se tuvo que asegurar que todos los sujetos se expusieran a los efectos de cada refactoring en la interfaz. Esto fue tenido en cuenta para el diseño de las tareas que los usuarios completaron durante los tests. Cada tarea se descompuso en pasos, y en cada paso los usuarios utilizaron una parte de la interfaz afectada por al menos un refactoring (en la versión refactorizada de cada sitio) y, por lo tanto, estos pasos pudieron utilizarse como entidades mesurables de las cuales se pudo analizar el impacto de los refactorings individualmente.

7.1.2. Preguntas de investigación, hipótesis, y métricas

El objetivo del experimento fue el de estudiar el efecto que cada refactoring tiene en la *calidad en uso*, específicamente la *usabilidad en uso*. Dado que ISO/IEC 25010 define *usabilidad en uso* como una medida de calidad de software compuesta de *efectividad en*

[18] Zencart - https://www.zen-cart.com
[19] WeBid - http://www.webidsupport.com

uso, *eficiencia en uso* y *satisfacción en uso*, las preguntas de investigación se definieron teniendo en cuenta estas tres métricas, para cada refactoring en particular:

- RQ1: ¿La efectividad en uso se ve afectada por el refactoring? Para responder esta pregunta se midió para cada refactoring el porcentaje de éxito obtenido por cada tarea completada (precisión y completitud). La hipótesis nula para abordar esta pregunta fue la siguiente:

$H_0$1: La efectividad en uso de un sistema luego de ser refactorizado es similar a la efectividad en uso del sistema antes de ser refactorizado.

- RQ2: ¿La eficiencia en uso se ve afectada por el refactoring? Para responder esta pregunta de investigación se midió, para cada uno de los refactorings, el tiempo que tardaron los usuarios en completar los pasos en los que está involucrado el refactoring en cuestión. La hipótesis nula para abordar esta pregunta de investigación es:

$H_0$2: La eficiencia en uso de un sistema luego de ser refactorizado es similar a la eficiencia en uso del sistema antes de ser refactorizado.

- RQ3: ¿La satisfacción se ve afectada por el refactoring? Para responder esta pregunta de investigación se midió para cada refactoring, las opiniones personales de los usuarios acerca de su satisfacción luego de completar tareas en las aplicaciones refactorizadas. La hipótesis nula para abordar esta pregunta de investigación es:

$H_0$3: La satisfacción en uso de un sistema luego de ser refactorizado es similar a la satisfacción en uso del sistema antes de ser refactorizado.

Las *variables de respuesta* son los valores medidos en el experimento para estudiar cómo los factores los influyen (Juristo and Moreno, 2010). La variable de respuesta *RQ1* es *efectividad en uso*, cuya métrica (*M1*) es el nivel de éxito (completado correcto) alcanzado durante la ejecución de un paso. M1 puede verse tomar el valor 1, que significa que el paso fue completado correctamente; o 0, que significa que el paso falló. Por ejemplo, si un refactoring estuvo involucrado en 4 actividades, 3 de las cuales se completaron exitosamente y 1 que no, el resultado promediado sería de 0.75. Esto,

naturalmente, resulta en valores entre 0 y 1 (incluyendo decimales) para medir la efectividad en uso de los refactorings compartidos entre diferentes pasos.

La variable de respuesta *RQ2* es *eficiencia en uso*, cuya métrica (*M2*) es el porcentaje de efectividad dividido por el tiempo empleado durante el paso. Para cada ejecución de un paso, se midió el tiempo de completado en segundos. Luego se calculó la proporción entre efectividad y tiempo. Si un refactoring afecta múltiples pasos, los valores de eficiencia para cada uno se agregaron mediante el cociente entre el promedio de efectividad por el promedio de tiempo (es decir, un promedio dividido por otro promedio). Los posibles valores para eficiencia son números positivos.

La variable de respuesta *RQ3* es *satisfacción en uso*, cuya métrica (*M3*) es una escala Likert de 5 puntos capturada mediante un cuestionario, en el que se pregunta a los usuarios por el nivel de acuerdo (desde “Totalmente de acuerdo” hasta “Totalmente en desacuerdo”) respecto de afirmaciones del estilo “*fue fácil buscar un producto específico del catálogo del sitio*”. Por lo tanto, los valores posibles para satisfacción en uso dentro de este experimento son valores entre 1 y 5.

El experimento tiene un diseño apareado bloqueado por objetos experimentales (Juristo and Moreno, 2010) dado que se tenían dos valores por cada variable de respuesta por sujeto (ver Tabla 2). Esto permitió trabajar con una base más amplia de muestras, ya que se evitó la división de sujetos por tratamiento para contrastarlos entre sí.

Tabla 2 Diseño del experimento

Combinaciones	Zencart (P1)		Webid(P2)	
	R	~R	R	~R
#1	1ro			2do
#2		1ro	2do	
#3		2do	1ro	
#4	2do			1ro

Para evitar el sesgo de aprendizaje, cada sujeto interactuó sólo con una versión de cada problema. De esta manera, cada tratamiento se aplicó sobre una sola versión de un

solo problema. Esto resulta en cuatro posibles combinaciones de sujetos que realizaron el experimento. La Tabla 2 detalla cada combinación. Se balancearon los sujetos asignados a cada grupo para bloquear el posible sesgo proveniente de comenzar siempre con el mismo problema o tratamiento. Es importante notar que las 4 combinaciones se aplicaron en las dos localizaciones (AR y SP).

7.1.3. Procedimiento

Para realizar el experimento, se creó una herramienta que guía a los usuarios por las tareas y sus pasos (en adelante será referida como *GuideTool*). La herramienta ayudó a determinar también en muchos casos si el paso ejecutado había sido completado con éxito o no (utilizado para la métrica M1), y midió los tiempos de cada uno (métrica M2), además de mostrar los formularios de satisfacción cuando fue necesario (métrica M3), almacenando las respuestas. La herramienta simplificó enormemente la mecánica de los experimentos, guiando a los usuarios a través de cada paso de las tareas y recolectando las métricas automáticamente.

La Figura 30 muestra la herramienta funcionando; se trata de una barra colocada en la parte inferior del navegador.

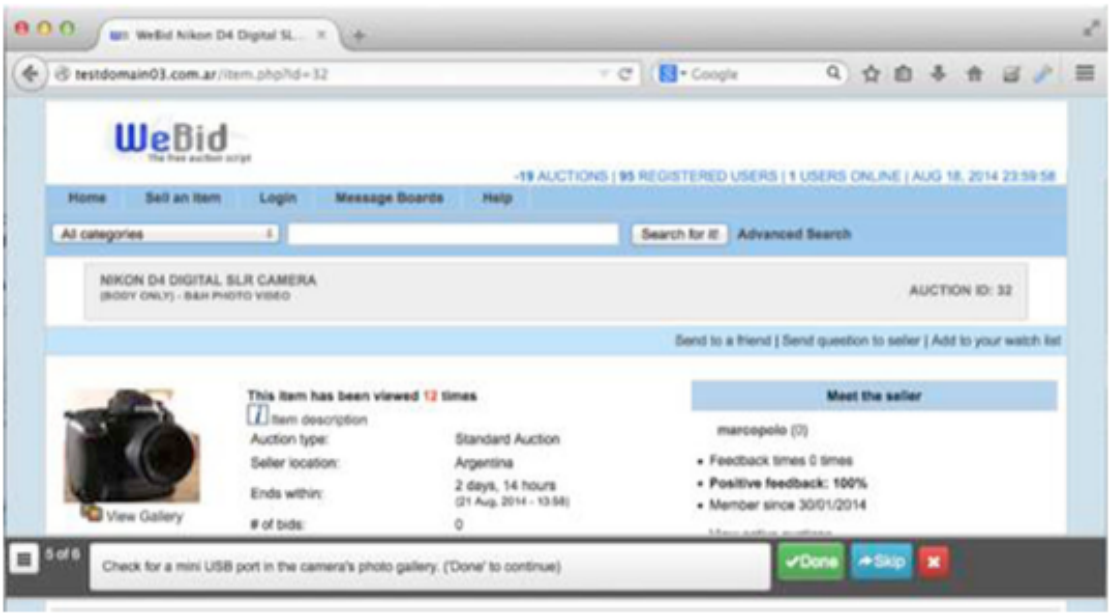

Figura 30. Herramienta GuideTool para conducir el experimento de refactorings y recolectar automáticamente la información de los voluntarios.

La herramienta permite a los usuarios ver el flujo completo de la tarea. Además, se indica el paso actual junto con el número total de pasos, una descripción de lo que se debe

hacer, y controles para indicar que el paso se ha finalizado (cuando no es posible determinarlo automáticamente), saltear el paso, o salir de la tarea por cualquier eventualidad.

Antes del experimento, se clasificó a cada sujeto dentro de uno de las cuatro combinaciones del diseño. Los grupos se crearon de manera de que queden lo más balanceados posible para cada combinación.

Los sujetos comenzaron completando sus respectivos cuestionarios de datos demográficos. Luego, despendiendo de la combinación asignada, comenzaron a interactuar con una de las dos aplicaciones (P1 o P2), ya sea con o sin refactorings. Para cada paso a completar, leyeron las instrucciones e intentaron completarlo sobre la aplicación. La herramienta guardó, de forma invisible para el usuario, el porcentaje de completitud y el tiempo empleado en cada paso. Estos valores se utilizaron para las métricas M1 y M2, para calcular la efectividad y eficiencia respectivamente. Una vez que el sujeto finalizaba un grupo de pasos (una tarea), se le presentaba el formulario de satisfacción para obtener la métrica M3. El proceso se repitió para todos los pasos restantes. Como se explicó anteriormente, los voluntarios tuvieron la posibilidad de saltear pasos que no pudieran completar.

Notar que todos los pasos y cuestionarios fueron equivalentes para ambas versiones de cada problema específico. De esta manera fue sencillo contrastar las métricas de efectividad, eficiencia y satisfacción.

Una vez que un sujeto finalizaba con la primera aplicación, el experimento continuaba con la segunda. Este segundo problema era, como se explicó en el diseño del experimento, uno diferente con el tratamiento opuesto al primero. El proceso en esta segunda fase fue idéntico al de la primera. Al final del procedimiento, para cada sujeto, se obtuvo información para una aplicación web con refactorings, y otro conjunto de valores para la otra aplicación sin refactorings.

7.1.4. Amenazas de sesgos

En esta sección se describen las amenazas de sesgos al experimento, de acuerdo a Wohlin et al. (Wohlin *et al.*, 2012), y qué acciones se tomaron para evitar o minimizar los efectos. Para paliar el efecto de la amenaza *bajo poder estadístico*, que aparece cuando

el trabajo se realiza con menos de 30 sujetos, se reclutaron 22 y 27 sujetos en cada replicación respectivamente. Se minimizó esta amenaza utilizando tests estadísticos suficientemente poderosos para tales cantidades de voluntarios: Wilcoxon para muestras repetidas, que es no-paramétrico. El uso de muestras repetidas se justifica dado que los tratamientos se aplican al mismo sujeto dos veces. Estos sujetos son las unidades experimentales dado que se observan los resultados de aplicar los tratamientos a través de ellos. La posibilidad de saltear tareas resulta en algunas medidas ausentes, lo que decrementa el número de datos a analizar. El experimento sufre de este sesgo sólo en los refactorings con métricas vacías para algunos sujetos. Se minimizó este efecto descartando los sujetos con más del 10% de pasos salteados. Todos los sujetos descartados correspondieron a la replicación SP, que estuvo inicialmente compuesta de 36 sujetos. *Confiabilidad de medidas*, que aparece cuando no se pueden confiar las medidas obtenidas, es otro posible sesgo. Se minimizó automatizando la obtención de métricas lo máximo posible. Las métricas de eficiencia y efectividad se calcularon con la GuideTool automáticamente para la mayoría de los casos, evitando errores humanos. La métrica de satisfacción es la única que sufre de esta amenaza, dado que no se puede asegurar que los sujetos no hayan cometido errores al completar el formulario. El cuestionario utilizado fue creado específicamente para este experimento, lo que resulta también en una falta de validación, dado que sólo se utilizó en este trabajo.

Adicionalmente, algunos refactorings fueron aplicados a más de un paso de las tareas del experimento. Para procesar la información, se promediaron todas las métricas por refactoring. En esta acción, se pueden haber perdido algunos resultados significativos. Por lo tanto, las métricas para efectividada, eficiencia y satisfacción sufren de este posible sesgo para los refactorings que aparecen en más de un paso (R2, R3, R4, R7, R11 en Zencart y R1, R2, R3, R4, R9, R15, R16 en Webid).

También se consideró la amenaza de sesgo de *Maduración*, que significa que los sujetos reaccionan diferente a medida que el tiempo pasa, se evitó limitando el experimento a 2 horas, lo cual es un tiempo más que suficiente para terminar todas las tareas. Los sujetos que pasaron más de este límite de tiempo no fueron considerados en el análisis. El sesgo *Selección*, que significa que un resultado obtenido de una base pequeña de voluntarios puede no ser representativo de la población, se abordó al replicar el experimento con sujetos de dos perfiles bien diferentes: sujetos que desarrollan

aplicaciones web y sujetos que interactúan con aplicaciones como usuarios finales exclusivamente. Además, las replicaciones se realizaron en dos países diferentes. *Aprehensión de evaluación*, que aparece cuando los sujetos temen ser evaluados. Esta amenaza se minimizó reclutando sólo voluntarios, y dejando lo más clara posible la premisa de que lo que se evalúa es el sistema (problema) y no ellos.

Se tuvieron en cuenta también sesgos de validez externa como *Interacción de selección y tratamiento*, que aparece cuando se tiene una población no representativa de la que se quiere generalizar. Se minimizó utilizando diferentes perfiles de usuarios en cada replicación. Otras amenazas se discuten en la publicación (Grigera *et al.*, 2016).

7.1.5. Análisis e interpretación

Dado que se realizó un diseño entre-sujetos que no sigue una distribución normal, se aplicó un test de Wilcoxon para muestras apareadas con un valor α=0.05. La hipótesis $H_0$1 anteriormente presentada indicaba: “La efectividad en uso de un sistema luego de ser refactorizado es similar a la efectividad en uso del sistema antes de ser refactorizado.”; $H_0$2, “La eficiencia en uso de un sistema luego de ser refactorizado es similar a la eficiencia en uso del sistema antes de ser refactorizado” y H03: “La satisfacción en uso de un sistema luego de ser refactorizado es similar a la satisfacción en uso del sistema antes de ser refactorizado.”. La Tabla 3 muestra los resultados del test para la hipótesis nula para cada refactoring (los 21 refactorings aplicados durante el experimento se identifican como R1 … R21). Un número 1 en la tabla significa que la versión refactorizada de la aplicación web obtuvo resultados significativamente mejores que permiten rechazar la hipótesis a favor de la versión refactorizada. Por otro lado, un -1 significa que la hipótesis nula fue rechazada a favor de la versión no refactorizada. Finalmente, un 0 significa que no hubo diferencias significativas para rechazar la hipótesis nula correspondiente. De los valores en la Tabla 3 se pueden derivar algunas conclusiones:

- Hay 11 refactorings que demostraron estadísticamente ser beneficiosos en aspectos de “usabilidad en uso” en al menos una replicación del experimento.
- El proceso de refactoring no muestra mejoras significativas en todos los casos, y un análisis más exhaustivo se requiere en cada situación.

- Hay más refactorings significativamente mejor puntuados en la replicación AR que en SP, posiblemente debido a las diferencias en los perfiles de usuario.
- No hay refactorings que muestren resultados negativos en satisfacción. Esto permite concluir que la experiencia de usuario mejora o al menos se preserva con los refactorings.

En el resto de este análisis categorizan los refactorings en tres grandes grupos: *beneficiosos* (11), *dudosos (o sin beneficio demostrable)* (6), y *no beneficiosos* (5). Es importante destacar que la suma de estos valores da 22 refactorings, aunque hay 21 refactorings en este experimento; esto sucede porque R3 pertenece a dos categorías al mismo tiempo: *beneficiosos* y *no beneficiosos*, dado que favorece tanto la versión refactorizada como no refactorizada en diferentes aspectos de cada replicación. Los refactorings beneficiosos se subclasifican por la replicación afectada (AR y SP).

Tabla 3. Resultados del experimento sobre refactorings. Los resultados favorable pueden verse resaltados.

# Ref	Replicación SP			Replicación AR		
	Efectividad	Eficiencia	Satisfacción	Efectividad	Eficiencia	Satisfacción
R1	0	0	0	1	1	1
R2	0	0	0	0	0	0
R3	-1	0	0	0	1	1
R4	0	0	0	0	-1	0
R5	0	0	0	0	0	0
R6	0	0	0	0	0	1
R7	0	-1	0	0	-1	0
R8	0	0	0	0	1	0
R9	0	0	0	0	1	0
R10	0	1	1	0	1	1
R11	0	0	0	0	0	0
R12	0	0	0	0	0	0
R13	0	0	0	0	0	0
R14	0	0	1	0	0	1
R15	0	0	0	0	0	1
R16	0	0	0	0	0	0
R17	1	0	1	0	0	0
R18	0	0	1	1	1	1
R19	0	0	0	0	-1	0
R20	0	0	1	0	0	0
R21	0	-1	0	0	0	0

Refactorings beneficiosos

Los resultados directos del test estadístico muestran que 11 de los 21 refactorings proveen un **beneficio claro**, esto es, sus resultados en el análisis rechazan al menos una

de las tres hipótesis nulas a favor de las versiones refactorizadas de las aplicaciones. Por lo tanto, se consideraron importantes para mejorar usabilidad y deberían tener una alta prioridad en el proceso de refactoring de usabilidad de una aplicación web.

Los refactorings que mostraron ser significativamente beneficiosos en ambas replicaciones del experimento son R10, R14 y R18.

R10) *Provide breadcrumbs:* la idea de este refactoring es ayudar a los usuarios a seguir el camino de navegación que siguieron hasta la página en la que se encuentran, permitiéndoles navegar rápidamente hacia los pasos anteriores.

R18) *Enable user to go back in the process steps:* este refactoring permite a los usuarios retornar a un paso previo durante un proceso. Sin este refactoring, un proceso sólo permite navegar hacia delante, forzando a los usuarios a comenzar de nuevo si desean corregir datos en un paso previo. El refactoring semi-automático *Turn Attribute Into Link* implementado en Kobold sirve en este caso cuando el proceso permite mantener el estado entre navegaciones.

R14) *Keep the user informed on the ongoing process:* este refactoring agrega información para mostrar el estado actual del proceso corriente, como los contenidos del carrito de compra y el precio total, o el estado de una oferta.

También se evaluaron los resultados separando por replicación. Los refactorings con resultados significativos en AR son R1, R3, R6, R8, R9 y R15:

R1) *Improve the description of process links:* este refactoring hace que los links sean más fáciles de comprender simplemente cambiando su texto o agregando íconos. En la replicación AR, este refactoring demostró beneficios en las tres variables de usabilidad en uso. En Kobold, puede usarse el refactoring equivalente *Rename Element*.

R3) *Anticipate a validation activity*: este refactoring agrega validación instantánea a un formulario, previniendo que los usuarios lo envíen sin la información completa, o correcta. El refactoring *Add Validation* de Kobold está basado en este refactoring.

R15) *Improve link destination announcement*: este refactoring se basa en el patrón “Link Destination Announcement” (Nanard, Nanard and Kahn, 1998), que propone enriquecer el contenido de un vínculo con widgets adicionales para presentar información adicional

sobre la página de destino. Esto ayuda a reducir navegaciones innecesarias. En Kobold, este refactoring puede aplicarse mediante *Rename Element* o incluso *Add Tooltip.*

R6) *Add a "confirm and commit" activity*: este refactoring agrega un resumen al final de las tareas con múltiples pasos, proveyendo a los usuarios de una oportunidad de revisar sus elecciones antes de enviar el formulario.

R8) *Introduce information on demand*: dado que este refactoring permite obtener información al pasar el mouse por un lugar específico, pero sin navegar, la mejora en eficiencia puede deberse al ahorro en tiempo. En Kobold puede usarse *Add Tooltip* para agregar ayudas contextuales.

R9) *Turn attribute into link*: de manera similar a *Introduce information on demand*, este refactoring ha demostrado mejoras significativas en eficiencia para la versión R, tal vez debido al ahorro de tiempo en búsqueda y navegación. Este refactoring está implementado en Kobold bajo el mismo nombre.

Los refactorings con mejoras significativas en SP son R17 y R20:

R17) *Make explicit the steps composing the process and the current step being executed*: este refactoring se utiliza durante el largo proceso de checkout en el sitio de e-commerce, y el proceso de publicación para el sitio de subastas.

R20) *Add a processing page*: este refactoring informa el progreso de un proceso largo que está siendo ejecutado. No hubo mejoras en eficiencia, lo que era en realidad esperado dado que el tiempo de proceso es igual en ambas versiones, tanto en R como en ~R; la única diferencia está en cómo el sistema informa al usuario lo que está sucediendo. Kobold es capaz de aplicar este refactoring de forma automática en muchos casos.

Refactorings dudosos

Hay seis refactorings en la Tabla 3 para los cuales no se pudo obtener suficiente evidencia para rechazar la hipótesis nula. Éstos son R2, R5, R11, R12 y R16.

Refactorings no beneficiosos

En la Tabla 3 hay cinco refactorings que muestran resultados negativos. Esto significa que la versión ~R obtuvo mejores resultados que la versión R en al menos un aspecto. Hay un solo refactoring que resultó ser ineficiente en ambas replicaciones: R7 (*Aggregate Activities*). Además, hay dos refactorings que resultaron ineficientes sólo en la replicación

AR: R4 (*Change Widget*) y R19 (*Add a summary activity*) y uno más que resultó ineficiente en SP: R21 (*Add an assistance activity*). Sólo el refactoring R3 (*Anticipate a validation activity*) mostró muy buenos resultados en AR en términos de eficiencia y satisfacción, pero resultó al mismo tiempo inefectivo en SP.

7.1.6. Discusión

Las secciones anteriores proveen un amplio espectro de resultados de los experimentos realizados en diferentes tipos de análisis. En resumen, los resultados muestran la situación particular en la cual los refactorings son más útiles en mejorar la usabilidad:

- Hay refactorings que siempre mejoran la usabilidad (R10, R14, R18);
- Hay refactorings que son más útiles para los usuarios con conocimiento en desarrollo de software (R1, R3, R6, R8, R9, R15);
- Otros refactorings son más útiles para aquellos sin conocimiento de desarrollo de software (R17, R20);
- Hay refactorings que son más útiles para aquellos que utilizan sitios de e-commerce frecuentemente (R3, R8, R9, R10, R14, R18, R21), mientras que otros refactorings son mejores para usuarios esporádicos (R5, R17, R19).
- Algunos refactorings son mejores para sitios de e-commerce (R4, R7, R11) y otros son mejores para sitios de subastas (R2, R3).

Para aplicar los refactorings se reclutaron 3 estudiantes de diferentes niveles y experiencia para cumplir el rol de desarrolladores. Se les pidió que ponderaran, ni bien terminaran de implementar cada refactoring, el esfuerzo invertido en cada implementación en categorías *fácil, medio* o *difícil*, y luego se promediaron los resultados.

De todos los análisis presentados, se pudo generar una lista priorizada de refactorings para que los desarrolladores puedan utilizar sistemáticamente para mejorar la usabilidad de sus sitios luego de haber sido diseñados. La estrategia para confeccionar la lista de prioridades fue la siguiente:

1ro) por número de significancias (cantidad de valores “1” en la Tabla 3);

2do) por número de diferentes variables dependientes que tuvieron significancia.

3ro) por nivel de esfuerzo en implementación, de fácil a difícil

4to) por tamaño de efecto

De esta estrategia se obtuvo la lista priorizada que se muestra en la Tabla 4.

Tabla 4. Lista de refactorings ordenada por prioridad

# Ref	Name
R10	Provide breadcrumbs
R18	Enable user to go back in the process steps
R1	Improve the description of process links
R17	Make explicit the steps composing the process and the current step being executed
R14	Keep the user informed on the ongoing process
R3	Anticipate a validation activity
R6	Add a “confirm and commit” activity
R15	Improve link destination announcement
R20	Add processing page
R8	Introduce information on demand
R9	Turn attribute into link
R2	Split page
R5	Add a verification activity
R11	Move widget
R4	Change widget
R21	Add an “assistance” activity
R16	Remove duplicated process links
R12	Split list
R19	Add a summary activity
R13	Distribute menu
R7	Aggregate activities

Esta lista puede servir para optimizar el esfuerzo de desarrollo, descartando refactorings insignificantes o difíciles de implementar.

7.2. Validación de Usability Events

El éxito de la metodología propuesta depende decididamente de la correcta detección de usability events. Algunos de ellos involucran para su detección heurísticas que pueden ser proclives al error, como interpretar la intención del usuario (*“¿quiso realmente el usuario hacer click en este elemento?”*) o inferir el significado de un elemento de interfaz (“*esto parece ser un formulario de búsqueda”*). Primero se realizó una evaluación preliminar con múltiples experimentos sobre diferentes sitios web, y gracias a eso se ajustaron las heurísticas de detección, incluyendo los diferentes umbrales y parámetros. Siguiendo esta evaluación previa, se organizó un experimento para validar la exactitud, precisión y

cobertura de la herramienta al capturar usability events mediante estas heurísticas configuradas con estos parámetros (Grigera *et al.*, 2017). Esta sección describe este experimento.

Para la validación se tuvieron que considerar dos escenarios muy diferentes. Por un lado, hubo que validar las heurísticas que involucran interpretar las intenciones del usuario, y por el otro, las que tienen que ver con inferir el propósito de un elemento DOM de interfaz. Otras heurísticas son suficientemente simples o no involucran ninguna interpretación subjetiva, con lo cual no fue necesario incluirlas en el experimento. A continuación, se detallarán los tres casos.

7.2.1. Definición y Planificación del experimento

Antes de planificar el experimento, se determinó que había 3 grupos de heurísticas entre los 12 usability events que puede detectar la herramienta:

1. **Heurísticas que involucran intención del usuario:** los eventos que dependen de este tipo de heurísticas son *Tooltip Attempt, Click Attempt, Flash Navigation, Navigation Path,* y *Flash Scrolling.*
2. **Heurísticas que involucran propósito de elementos DOM:** los eventos en este grupo son: *Bulk Action, Search,* y *Form Submission*.
3. **Heurísticas de bajo nivel:** estas heurísticas dependen exclusivamente de eventos de bajo nivel que, si bien podrían fallar, esto sucedería en escenarios que son marginales e incluso difíciles de reproducir en un conjunto limitado de aplicaciones web de prueba. Por este motivo se excluyeron de la validación. Son los siguientes:
    a. **Text Input:** completar un campo de texto es un evento de muy bajo nivel y extremadamente difícil de fallar en su captura (falso negativo) o que el usuario lo haga por accidente (falso positivo).
    b. Option Selection: este evento detecta cuándo un usuario ha seleccionado una opción de un select box o un conjunto de radio buttons. No se validó dado que los casos en los que se utilizan elementos no-estándar, si bien existen, son muy inusuales.

c. **Long Request:** este evento en particular detecta un tiempo largo entre requests. Luego de 3 segundos entre cargas de página, el evento se dispara. Si bien no se descarta una detección fallida, en los tests preliminares no se hallaron casos sin detectar. Es importante tener en cuenta que el umbral de 3 segundos se ajustó por el sólo hecho de no descartar ningún evento, y que el usability smell asociado tiene un umbral mucho mayor, de 10 segundos, como se explicó en el capítulo 4.

d. **Unfilled Form:** este evento combina la lógica de detección de Text Input, para interpretar cuándo un usuario comienza a completar un formulario, con el evento JavaScript de bajo nivel *page unload*, que puede ser capturado sin problemas en la vasta mayoría de aplicaciones web. Por lo tanto, tiene una probabilidad marginal de no ser detectado.

Para validar las heurísticas del grupo 1, se pidió a un grupo de voluntarios que llevaran a cabo interacciones específicas, y luego se compararon los eventos de usabilidad capturados por la herramienta con la observación directa, "manual" de dichas interacciones. El experimento se preparó creando escenarios donde estos eventos de usabilidad deberían ser detectados, y con un segundo conjunto de escenarios donde los eventos *no* deberían ser detectados, pero la herramienta podría ser "engañada", detectando un falso positivo. Se crearon diferentes grupos de estos escenarios para generar tareas más significativas, con el objeto de recrear situaciones realistas para los voluntarios. Se utilizaron 3 sitios web diferentes de distintos dominios de negocio, y se reclutaron 15 voluntarios para correr las tareas en 2 sitios cada uno. Los voluntarios fueron 8 hombres y 7 mujeres, con edades entre 26 y 64 años ($\bar{x}$ 35.66, s^2 75.52), y diferentes formaciones y niveles de conocimiento de los sitios, desde visitantes regulares hasta novatos.

Durante el experimento, se dejó la herramienta corriendo para que capturara usability events, mientras se observó cuidadosamente las interacciones llevadas a cabo por los usuarios para determinar qué eventos estaban produciéndose, y cuáles podrían estar siendo incorrectamente registrados. Para esto, se involucró a dos evaluadores: mientras uno guiaba al voluntario a través de las tareas, el otro monitoreaba los eventos al instante en que sucedían utilizando una versión modificada de Kobold que provee una

visualización inmediata de los mismos. Mientras el primero estaba a cargo de buscar los eventos no detectados (falsos negativos), el segundo se concentró en detectar los que estaban incorrectamente capturados (falsos positivos). En los casos en los que se utilizaron sitios web populares, donde naturalmente no se puede instalar el script, se incorporó de todas maneras usando extensiones de navegador.

Las heurísticas en el grupo 2 (para los eventos *Form Submission*, *Search* y *Bulk Action*) están basadas en interpretar elementos DOM, con lo cual no fue necesario reclutar voluntarios, dado que su interpretación no está en juego. En cambio, estas heurísticas se validaron a través de diferentes sitios web para comprobar la capacidad de la herramienta de detectar la misma situación en contextos diferentes. Los sitios evaluados fueron obtenidos del ranking de sitios más visitados de Alexa[20] (tanto locales como globales), más aquellos utilizados en el experimento de las heurísticas del grupo 1. Se aplicaron dos criterios para seleccionar los sitios: el primero fue asegurar que los sitios involucraran la interacción necesaria. El segundo criterio fue puramente técnico: debido a la tecnología empleada para simular la instalación del script, parte de la herramienta fue bloqueada por los sitios por motivos de seguridad, por lo cual se tuvo que descartar estos sitios.

7.2.2. Resultados

Luego de haber realizado el experimento con ambos grupos de heurísticas, se examinaron los resultados observados, y en el caso del grupo 1, también los usability events registrados por la herramienta. Se empleó un score F2 para obtener métricas de precisión y cobertura (*precision and recall*) del funcionamiento de la herramienta. Se decidió por este tipo de score (en contraste con el F1) porque da más relevancia a la cobertura que a la precisión, y en esta etapa del proceso se prioriza la cobertura dado que durante la detección de usability smells siempre se pueden descartar falsos positivos, pero es imposible recuperar los eventos que no fueron capturados. La Tabla 5 muestra todos los valores obtenidos por cada usability event.

[20] www.alexa.com accedido el 8/6/2016 – actualmente ya no ofrece el ranking completo de forma gratuita

Tabla 5. Resultados del experimento sobre captura de usability events

	Exactitud	Cobertura	Precisión	Score F1	Score F2
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

La **exactitud** (*accuracy*) determina el porcentaje de eventos correctamente diagnosticados por la herramienta, es decir correctamente capturados u omitidos, según el criterio establecido por la observación directa de los voluntarios en el caso del grupo 1, o el criterio establecido manualmente para los elementos DOM en el grupo 2. La **cobertura** (*recall*) indica el número de eventos correctamente capturados con respecto a todos los que *deberían haber sido* capturados según los criterios de referencia, sin considerar omisiones (ya sean éstas correctas o no). Finalmente, la precisión (*precision*) muestra el nivel de falsos positivos, es decir, de todos los eventos capturados, la proporción de correctos sobre incorrectos. Los scores F1 y F2 representan promedios ponderados de estas medidas:

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

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

Como se explicó anteriormente, se dará más importancia al score F2 dado que es más importante destacar la cobertura.

7.2.3. Discusión

Ciertos resultados merecen una explicación. Los altos porcentajes de cobertura en los eventos *Click Attempt* y *Flash Scroll* significan que es raro que la herramienta omita la captura, lo cual resulta coherente con las lógicas de sus heurísticas. Las omisiones, sin

embargo, pueden ocurrir. Algunos ejemplos de capturas fallidas de *Click Attempt* son clicks hechos por selección de texto interpretados como clicks intencionales (si bien el script está preparado para ignorarlos, puede fallar si el usuario simplemente no deja presionado el botón suficiente tiempo), o clicks hechos específicamente *luego* de una selección de texto que no se capturan con la heurística actual. En el caso de *Flash Scroll*, el umbral está basado en la velocidad del scroll, y sencillamente falló en algunos casos en los que durante la observación se consideraron positivos. En las pruebas preliminares se determinó que un umbral más bajo capturaría demasiados falsos positivos, y si bien se prioriza la cobertura, en este caso la abundancia de falsos positivos sería perjudicial para el diagnóstico de usability smells. Otros umbrales similares basados en métricas de velocidad (en rigor, tiempos de espera) pueden sufrir el mismo efecto: *Tooltip Attempt*, *Navigation Path* y *Flash Navigation*. Sin embargo, estos umbrales son muy conservadores. En el caso de los eventos del grupo 2, *Bulk Action*, *Search* y *Form Submission*, la causa principal de eventos omitidos fue la falla al detectar correctamente los formularios, como interfaces HTML no estandarizadas, o acciones relacionadas a éstos, como el envío.

La captura de eventos también puede arrojar falsos positivos, que son consecuencia de la incorrecta interpretación de las intenciones del usuario, y por eso los scores F2 pueden bajar en comparación con la cobertura.

Para todos los eventos, los resultados de score F2 muestran valores relativamente alto (por encima de 0.65) en la performance de la herramienta. No obstante, es posible identificar, entre los scores F2, dos grupos de usability events. El grupo que mejores puntajes obtuvo ($0.75 < F2 < 0.97$ para *Click Attempt*, *Tooltip Attempt*, *Flash Scroll*, *Navigation Path*, *Bulk Action*) coincide mayormente con los eventos de usabilidad que dependen de la interpretación del usuario. El grupo con resultados más moderados ($.67 < F2 < .75$, *Search*, *Flash Navigation*, *Form Submission*) se corresponde mayormente con los eventos que involucran razonamiento sobre estructuras de elementos DOM.

Aún si las medidas F2 son aceptables para todos los eventos, mejorar las lógicas de aquellos que involucran análisis de elementos DOM incrementaría la cobertura en la captura de usability smells.

Los resultados pueden considerarse positivos, dado que la interpretación de elementos DOM es más sencilla de mejorar que la interpretación del usuario, simplemente analizando más casos de elementos HTML.

7.2.4. Riesgos de Invalidez

Hay un número de amenazas a la validez del experimento. Dado que se lanzaron dos validaciones diferentes, una con múltiples voluntarios y otra con múltiples sitios, se considerarán dos conjuntos separados de amenazas. En el experimento que involucró voluntarios, se descubrieron varias amenazas que se intentaron mitigar. Dado que hubo que buscar detección de eventos en sesiones breves, el experimento fue afectado por la amenaza conocida como **fishing**, es decir, buscar deliberadamente un resultado específico. Esto se redujo mediante el diseño de tareas significativas para los usuarios, para evitar que los evaluadores los llevaran por un camino específico que fuerce la generación de ciertos eventos sin un contexto razonable. Las amenazas **externas** fueron reducidas mediante la selección consciente de una muestra heterogénea de voluntarios, con diversas formaciones y experiencias previas con los sitios evaluados. Además, los sitios evaluados son reales, y si bien los voluntarios fueron provistos de los dispositivos para realizar las tareas, no se los movió de sus propios ámbitos laborales o domésticos. En el caso de las **amenazas sociales a la validez estructural**, que puede afectar los resultados cuando los usuarios temen ser evaluados ellos mismos, o simplemente actúan diferente de cómo lo harían fuera del contexto del experimento, se minimizó reclutando sólo voluntarios que no conocían lo que estaba siendo evaluado, cosa que es posible ya que la herramienta es invisible para ellos. Además, el proceso de captura no se ve afectado de forma positiva ni negativa por el comportamiento temeroso del voluntario.

Con respecto al experimento que involucra múltiples sitios, para las heurísticas del grupo 2, consideramos mayormente amenazas externas relacionadas a la representatividad de la muestra. Para paliar esto, se seleccionaron sitios de un ranking estándar de sitios populares (Alexa).

7.3. Validación de Usability Smells

Para evaluar la detección automatizada de usability smells, se analizó el comportamiento de la herramienta en sitios reales, ya sea de forma independiente, o en comparación con

hallazgos realizados por estudios de usuario con expertos en un ambiente controlado (Grigera *et al*., 2017). El método empleado para este experimento fue Goal–Question–Metric (GQM) (Basili, Caldiera and Rombach, 1994). GQM propone la definición de metas (*goals*), refinarlas en preguntas (*questions*) y especificar las métricas (*metrics*) requeridas para responder estas preguntas.

7.3.1. Preparación

La meta definida fue hallar el grado en el que la herramienta puede complementar o reemplazar los métodos manuales de evaluación de usabilidad. Para este propósito, se midió la confiabilidad de la herramienta, el grado de acuerdo con tests de usuario manuales, y los recursos que precisa.

La meta fue refinada en tres preguntas, con sus correspondientes métricas:

Q1: ¿Cuántos problemas de usabilidad puede hallar Kobold con respecto a un test de usuario manual?

M1: Número de problemas encontrados por cada método

M2: Porcentaje de problemas encontrados por Kobold respecto al test de usuario tradicional.

Q2: ¿Cuánto tiempo y recursos requiere Kobold respecto de un test de usuario manual?

M3: Horas-hombre empleadas en la detección de problemas.

M4: Recursos requeridos

Q3: ¿Cuán confiables son los resultados de Kobold?

M5: Número de resultados correctos.

M6: Número de falsos positivos.

De manera similar al experimento descripto en la sección anterior, se realizaron evaluaciones preliminares sobre sitios web diferentes de los evaluados en el experimento real, para ajustar los parámetros y umbrales de los buscadores de usability smells, de modo de bajar la tasa de falsos positivos lo máximo posible. La presencia de falsos positivos es la mayor amenaza que enfrenta la metodología, al ser una manera en

extremo sensible de hallar usability smells, con lo cual el ajuste de parámetros es debido mayormente a este punto. Se halló que, para obtener resultados consistentes, los parámetros requieren diferentes valores según el tráfico de los sitios. Las pruebas preliminares se llevaron a cabo en 2 períodos de 1 mes cada uno.

Luego de las evaluaciones preliminares, la primera parte del experimento consistió en correr tests de usuario manuales en las aplicaciones evaluadas. Para usar de referencia, se realizaron tests de usuario con 9 sujetos, 4 hombres y 5 mujeres con edades entre 25 y 62 años ($\bar{x}$ 35, s^2 125.25). Un conocido estudio de Nielsen afirma que mediante un solo test con 5 voluntarios se pueden hallar 85% de los problemas presentes en un sitio web[21]. Según este mismo estudio, el porcentaje de problemas hallados con *n* usuarios es $(1-(1- L)^n)$, donde L es la proporción de problemas de usabilidad descubiertos durante un test con un solo usuario. Un valor típico para L es 31%, basado en la experiencia de los autores. Utilizando esta fórmula, un solo test con 9 usuarios detectaría cerca de 96% de los problemas, lo cual lo hace un punto de referencia efectivo.

El experimento se realizó sobre dos aplicaciones reales: una agencia de viajes online (a la cual nos referiremos como aplicación TA), y un sitio de e-commerce de productos oficiales de un club de fútbol argentino de primera división (en adelante, aplicación FF). La selección de voluntarios para la aplicación TA se basó en estadísticas del Ministerio de Turismo[22] de Argentina, mientras que para el sitio FF se basó en una entrevista personal con el administrador del local al público de la misma franquicia. Observando la distribución demográfica, se reclutaron voluntarios que la representaran de manera proporcional. Se trabajó en conjunto con los administradores de los sitios en cada caso para diseñar las tareas, para que sean representativas de lo que los usuarios hacen corrientemente durante sus visitas.

Los voluntarios tuvieron que completar 5 tareas típicas en cada aplicación, mientras se observó su comportamiento y se midieron los tiempos. Para que se sintieran cómodos ingresando sus datos personales y realizando compras de una manera realista, se generaron clones de las aplicaciones reales. Luego de los tests, se obtuvo una lista de problemas de usabilidad para ambas aplicaciones (de manera de obtener valores para las

[21] http://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/
[22] http://www.turismo.gob.ar/estadisticas

métricas M1, M2, M5 y M6) y también los tiempos y recursos empleados por los desarrolladores y el personal (para las métricas M3 y M4).

La segunda parte del experimento consistió en recolectar resultados de Kobold. Para obtenerlos, se creó una cuenta para cada sitio, esta vez la versión real de producción, y se instaló el script tal como lo indican las instrucciones de la herramienta. Esto permitió medir el tiempo de instalación (para la métrica M3), y los recursos consumidos (para M4). La herramienta recolectó y analizó datos de interacción por un período de 5 semanas, 3 semanas en febrero de 2015 y 2 adicionales en mayo de 2016 (para M1, M2, M5 y M6). Durante estos períodos, se empleó una instancia de la herramienta corriendo en un VSP (Virtual Private Server – Servidor Privado Virtual) con 1gb de RAM y el sistema operativo Ubuntu Server 14.04 LTS.

7.3.2. Resultados

Respecto al a métrica M1, el número total combinado de problemas hallados por las dos metodologías fue de 27. De este total, habiendo procesado 52,568 eventos (25,556 de la aplicaciónFF y 27,012 de la aplicación TA), Kobold encontró 18 smells **(66.67% del total de 27)**, mientras que mediante el test manual se hallaron 19 **(70.37%)**, con una intersección de 10 problemas **(37.04%)**. Respecto a esta intersección, si sólo consideramos los 19 problemas hallados manualmente, la herramienta encontró **52.63%**.

Entre los problemas que ambos métodos hallaron, el formulario de registración a la aplicación TA no señalaba los campos obligatorios, cosa que se observó durante los tests manuales, y Kobold reportó con el smell *Late Validation*. En la aplicación FF, los usuarios querían retornar a los pasos previos durante el proceso de compra (o *checkout*), pero los títulos indicadores de los pasos eran de sólo lectura; esto se encontró tanto en los tests manuales como en la herramienta bajo el smell *Unresponsive Element*.

Respecto a los problemas exclusivamente hallados por los tests manuales y no por la herramienta, se encontraron comportamientos específicos de cada aplicación o inconvenientes detectados por lenguaje corporal, que son o bien muy difíciles de capturar, o no se pueden capturar automáticamente en absoluto. Por ejemplo, en el sitio FF, los usuarios se confundían constantemente con la acción “Agregar al Carrito”, que no mostraba claramente que el producto había sido agregado.

Por otra parte, los problemas hallados exclusivamente por la herramienta y no en los tests manuales fueron principalmente problemas relacionados con interacciones que no estaban cubiertas por las tareas diseñadas. Por ejemplo, en la aplicación FF, la herramienta detectó que muchos usuarios hacían click en un cartel de bienvenida que no estaba mencionado en las tareas, probablemente esperando que los lleve a una sección en particular.

Con respecto a las métricas M5 y M6 para los resultados de Kobold, el número total de problemas hallados fue en realidad más alto que los 18 válidos. La herramienta encontró 6 usability smells adicionales que resultaron en falsos positivos, lo que representa un 75% de verdaderos positivos (M5) y 25% de falsos positivos (M6). Como un ejemplo de falso positivo, un *Late Validation* fue hallado en la aplicación FF para el botón de “Agregar al Carrito”, que en realidad era un botón de envío o *submit* para un formulario. Luego del envío, el mismo formulario reaparecía, dado que agregar un producto al carrito no navegaba hacia el mismo, sino que el usuario permanecía en la página del producto. Esto se interpretó entonces como un envío de formulario fallido, cuando en realidad el producto estaba siendo agregado al carrito (cabe destacar que este fue de hecho un comportamiento confuso para los usuarios capturado en los tests manuales, como se explicó anteriormente). La Tabla 6 muestra los usability smells detectados separados por tipo, junto con la tasa de falsos y verdaderos positivos.

Tabla 6. Resultados del experimento sobre captura de usability smells

	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**

Respecto a la métrica M3 (horas-hombre), el tiempo total consumido por los tests manuales fue **de 5h 40' para TA y 6h 20' para FF**, empleado en diseño de tareas, generación de aplicaciones clon, tests de usuarios y análisis – sin contar los tiempos de viaje, mientras que la herramienta requirió **45' para FF y 35' para TA**, para instalar la herramienta y luego inspeccionar los resultados. Respecto a los recursos, la métrica M4, los tests manuales requirieron de 9 voluntarios para cada aplicación, dos expertos en usabilidad para realizar las evaluaciones y un desarrollador web para generar las aplicaciones de prueba (clones) en un servidor aparte. Kobold requirió un desarrollador web capaz de agregar el script de código en las aplicaciones evaluadas y el servidor en el cual corre la aplicación en sí, lo cual no representa en realidad un recurso para el usuario.

7.3.3. Discusión

La primera pregunta que se intentó responder respecto a la definición del experimento fue Q1: "*¿Cuántos problemas de usabilidad puede hallar Kobold con respecto a un test de usuario manual?*". El experimento demostró que la herramienta encontró la mitad de los problemas que pudo encontrarse con un test manual, lo cual es un resultado positivo, dado que significa que Kobold puede encontrar problemas similares a los de una técnica probada y establecida, pero en forma automática. Sin embargo, algunos de los problemas detectados sólo en los tests manuales (4 de 27) fueron demasiado específicos de cada aplicación, y requieren razonamiento humano, lo cual significa que Kobold probablemente nunca podrá hallarlos. No obstante, la herramienta encontró problemas que los tests manuales no pudieron hallar. Esto sucedió por dos motivos: por un lado, las tareas definen un curso fijo de acciones para los voluntarios, lo cual limita los problemas de usabilidad que éstos podrían encontrarse, y por otro lado, el límite de 9 usuarios (y por lo tanto la supuesta cobertura del 96%) también restringe el número de problemas que el experto puede encontrar. Kobold, siendo una herramienta que toma como entrada la interacción real de una gran masa de usuarios, pudo encontrar otros resultados.

Respecto a Q3: "*¿Cuán confiables son los resultados de Kobold?*", aunque la tasa de falsos positivos fue considerable (25%), pudiéndose mejorar, esto no hace a la herramienta ni al método poco confiables. Además, los resultados fallidos pueden arreglarse en la mayoría de los casos trabajando sobre las heurísticas de los buscadores individuales de usability smells.

Finalmente, se considera que Q2 "*¿Cuánto tiempo y recursos requiere Kobold respecto de un test de usuario manual?*" muestra un punto a favor de Kobold. Si bien se requiere de tiempo para conseguir resultados substanciales, dependiendo del tráfico del sitio, el costo y esfuerzo es mínimo, con lo cual existe un inmenso valor costo/beneficio para aquellos que no tienen recursos, o carecen del conocimiento para realizar un test de usuario.

Al responder estas tres preguntas, se consiguió el objetivo del experimento, es decir, se obtuvo una primera perspectiva del nivel en el cual el método puede reemplazar las evaluaciones manuales de usabilidad. La respuesta tiene dos aspectos. Primero, los resultados indican que la herramienta puede encontrar usability smells relevantes, y comparado con los problemas hallados por medio de tests manuales, aunque no puede reemplazar el nivel de razonamiento humano requerido para algunos de ellos. Segundo, el costo de analizar una aplicación web es extremadamente bajo, comparado con los tests manuales. Por ende, la herramienta puede servir como (1) una alternativa con esfuerzo casi nulo para desarrolladores sin recursos para realizar tests de usabilidad, y también (2) un buen complemento para los tests manuales, dado que puede hallar problemas que no siempre están cubiertos por las tareas de un test de usuario o evaluaciones heurísticas.

Otro aspecto positivo del experimento fue la oportunidad de agregar nuevos usability smells a la herramienta, dado que los tests manuales proveyeron nuevos datos. Por ejemplo, un problema notable hallado durante los tests en la aplicación TA fue un caso de **banner blindness**, que puede traducirse como "ceguera selectiva ante anuncios": los usuarios tenían que visitar una sección específica del sitio, pero dado que el link hacia ella parecía un anuncio (y no lo era) ni un solo voluntario lo encontró, a pesar de ser un enorme botón en el centro de la página principal. Este comportamiento puede ser generalizado para ser detectado como un usability smell.

7.3.4. Riesgos de Invalidez

El experimento fue por supuesto afectado por amenazas a la validez que requirieron de trabajo adicional. La más importante tal vez sea la **validez interna**, que fue abordada desde el diseño del experimento. Para mitigar **amenazas de instrumentación** respecto al entorno de los usuarios, se les permitió utilizar sus propios dispositivos en sus propios ambientes domésticos o laborales. Respecto a la **validez externa**, se buscó la

representatividad de la muestra para cada aplicación, como se describe en la preparación del experimento. Fue también un punto importante la amenaza de f**ishing**, que puede comprometer al experimento de dos maneras: correr el experimento manual conociendo los resultados de Kobold, o viceversa, es decir, conocer los resultados de los tests manuales antes de ajustar la herramienta para la detección. Para el primer caso, no se inspeccionaron los resultados hasta haber finalizado con los tests manuales, para impedir que los resultados de los últimos queden sesgados por el propio conocimiento de los hallazgos automáticos. También se redujo esta amenaza desde el diseño de tareas normales, con significado para el usuario como criterio principal, sin agregar más indicaciones que las necesarias. Para el caso opuesto, se configuraron los parámetros para los buscadores de usability smells en evaluaciones preliminares anteriores a los tests de usuario. Con respecto a la **validez de construcción**, se minimizaron los riesgos de aprehensión reclutando sólo voluntarios.

7.4. Validación de Refactorings Automatizados

En esta sección, se describe una primera validación sobre usability refactorings en un experimento con aplicaciones reales, basado en el experimento explicado en la sección previa sobre usability smells (Grigera, Garrido and Rossi, 2017).

7.4.1. Objetivo

La validación consistió en realizar tests de usuario sobre las aplicaciones analizadas en la sección 7.3: la agencia de viajes online (aplicación TA) y el sitio de e-commerce de un club de fútbol (aplicación FF).

Durante este nuevo experimento se aplicaron refactorings para resolver los usability smells hallados en el experimento previo. Los refactorings se aplicaron mediante la herramienta Kobold. En particular se aplicaron instancias de *Add Autocomplete*, *Add Validation*, *Turn Attribute into Link*, *Add Default Value* y *Add Processing Page*. La idea del experimento fue la de comparar los niveles de usabilidad sobre las aplicaciones antes y después de refactorizar.

7.4.2. Preparación

Para el experimento se reclutaron 8 voluntarios, 5 hombres y 3 mujeres (edades: $\bar{x}$ 37.5, s 9.74). Los voluntarios realizaron tareas en ambas versiones (es decir, con y sin refactoring) de las dos aplicaciones:

Sitio FF

1. Encontrar el precio para un producto específico
2. Agregar el producto al carrito y volver a la página de inicio.
3. Registrarse en el sitio.
4. Buscar un producto específico.
5. Completar la compra (*Checkout*)

Sitio TA

1. Buscar un paquete a un destino especificado.
2. Reservar el más barato
3. Buscar un aéreo hacia un destino y en una fecha especificados.

Se midieron tres aspectos relacionados a la usabilidad de acuerdo con el estándar ISO/IEC 25010: efectividad en términos de porcentaje de completado, eficiencia en tiempo promedio de completado, y satisfacción por medio de un cuestionario SUS estándar (Brooke, 1996).

Para evitar el sesgo de efecto de aprendizaje (*learning effect*), cada sujeto interactuó con una versión diferente de cada aplicación. Esto quiere decir que, si un sujeto dado realizó tareas sobre la versión original de FF, entonces trabajó con la versión refactorizada de TA, y nunca usó el mismo sistema dos veces. De esta manera, se obtuvieron 8 muestras para cada sitio: 4 para cada versión.

7.4.1. Resultados

Los resultados obtenidos muestran mejoras en todos los aspectos, de lo cual se desprende que la metodología, implementada mediante la herramienta Kobold, es de hecho capaz de mejorar la usabilidad de una aplicación web automáticamente. Los resultados detallados se muestran en la Tabla 7.

Tabla 7. Resultados del experimento sobre captura de usability smells

	Aplicación FF		Aplicación TA	
	Original	Refactorizado	Original	Refactorizado
Satisfacción	74.37	78.75	84.37	93.75
Eficiencia	03’40’’	02’57’’	03’09’’	02’36’’
Efectividad	100%	100%	83.33%	100%

Si bien el experimento es algo acotado, y es necesario realizar una versión extendida agregando más sujetos y sumando cobertura para todos los refactorings, los resultados son prometedores.

Todas las métricas fueron mejoradas en la versión refactorizada, excepto por un resultado: la efectividad de la aplicación FF fue óptima en ambas versiones. Luego, la satisfacción fue mayor en ambas aplicaciones, la eficiencia también, ya que los tiempos promediados fueron menores en ambos casos, y también se observó efectividad perfecta en el caso de la aplicación TA, cuando en la versión no refactorizada algunos usuarios tuvieron problemas para completar ciertas tareas.

Cabe destacar que, durante las pruebas, la carga dinámica de los refactorings no afectó la experiencia de los usuarios, que no notaron el procesamiento extra.

7.5. Validación de la Estrategia Scoring Map de agrupamiento de Eventos

Para validar la estrategia de agrupamiento basada en el algoritmo de similitud de elementos DOM (Zanotti, 2016) utilizada en muchos de los buscadores de usability smells se condujo un experimento con elementos DOM de sitios reales, donde se comparó agrupaciones realizadas a mano con los grupos generados automáticamente por esta estrategia. Se utilizaron otras estrategias conocidas publicadas en la literatura como punto de referencia.

7.5.1. Preparación

Para realizar el experimento se requirieron de dos piezas fundamentales: por un lado, un grupo de elementos DOM suficientemente grande obtenido de sitios reales. Además, estos elementos tuvieron que estar a su vez agrupados por afinidad. Esto se realizó con la asistencia de una herramienta programada para tal fin, arbitrando los grupos con tres evaluadores independientes. El conjunto obtenido contó con 868 elementos DOM agrupados en 135 grupos. Se procuró obtener grupos de elementos de diversos tamaños, con alturas (calculadas como altura de árbol – distancia entre la raíz y el nodo hoja más lejano) entre 1 y 7.

La segunda pieza de la preparación fue la implementación de los algoritmos de similitud de referencia. Para esto se crearon versiones de RTDM (Reis *et al.*, 2004), Bag of XPaths (Joshi *et al.*, 2003) y un algoritmo de comparación simple de XPaths.

7.5.2. Procedimiento

El procedimiento consistió en tomar los elementos DOM del conjunto de prueba y ejecutar los 4 algoritmos para que realizaran los agrupamientos. Luego se compararon estos grupos con el agrupamiento de referencia realizado manualmente. Para esta comparación se realizó un estudio de precisión y cobertura (precisión & recall), pero dado que la comparación que se realizó no se ajusta exactamente a este tipo de estudios, se analizaron los elementos en tuplas: si 2 elementos están juntos en el agrupamiento evaluado y también en el de referencia, se considera un verdadero positivo. Si sólo están en el agrupamiento evaluado, se considera un falso positivo (y de forma análoga con los negativos). Como número de referencia para la evaluación de cada algoritmo, se utilizó el score F2, que es un promedio entre precisión y cobertura (ver sección 7.2.2).

7.5.3. Resultados

Los resultados obtenidos (ver Figura 31) muestran una ventaja del algoritmo Scoring Map sobre todos los algoritmos de referencia, tanto en precisión (*precision)* como en score F2.

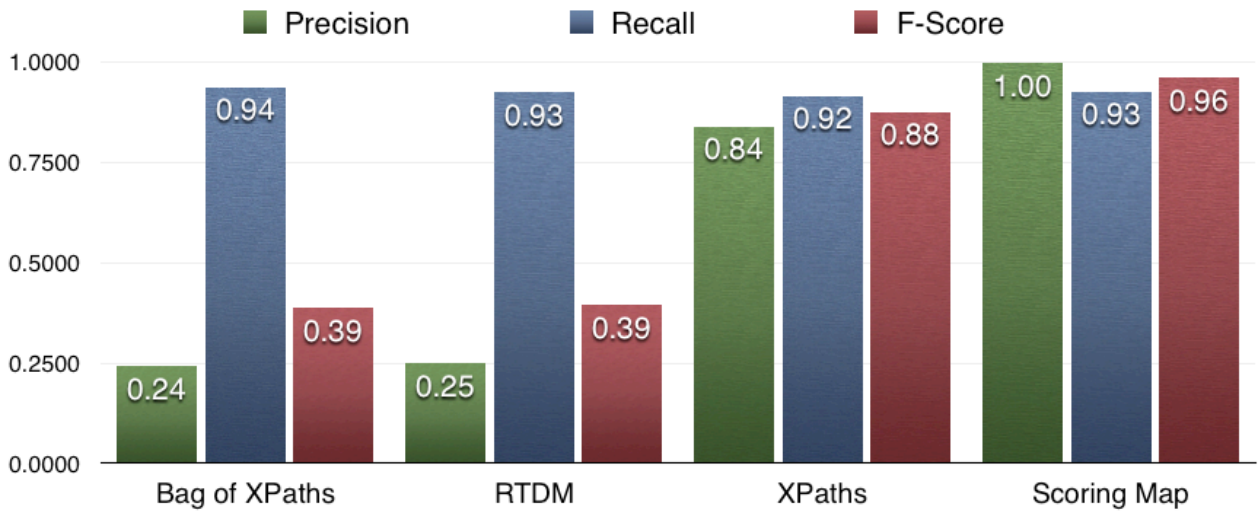

Figura 31. Resultados del experimento sobre el algoritmo Scoring Map.

La cobertura (*recall*) fue más uniforme, e incluso mayor en *Bag of Paths*. Esto de todas maneras significa simplemente que el algoritmo es muy permisivo, a menos que venga acompañado de un coeficiente alto también en precisión, que sólo es el caso en *Xpaths* y el algoritmo Scoring Map.

Capítulo 8. Conclusiones

En este trabajo se presentó una metodología para hallar problemas de usabilidad en aplicaciones web automáticamente, requiriendo el mínimo esfuerzo posible por parte de los desarrolladores. También se mostró que es posible, en algunos casos, ofrecer soluciones automatizadas para estos problemas utilizando la técnica de refactoring. Como prueba de concepto, se presentó **Kobold**, una herramienta capaz de poner en práctica todas las ideas de la metodología propuesta, y se realizaron diferentes validaciones para comprobar la utilidad de la misma.

Si bien el rango de problemas que se pudo hallar está limitado a lo que puede automatizarse, la metodología ofrece múltiples ventajas: en primer lugar, el esfuerzo de instalación es realmente insignificante. En la herramienta implementada, sólo es necesario completar un formulario de registro y pegar un *snippet* de código en la aplicación bajo análisis. Esto baja la barrera de la adopción al mínimo posible.

8.1. Resumen de Contribuciones

En este trabajo se realizaron las siguientes contribuciones:

- Una estrategia para procesar eventos de interacción de usuarios

Como se explicó en el Capítulo 4, el procesamiento de eventos realizado en el cliente facilita la detección posterior en gran medida, gracias al nivel de abstracción logrado en el pre-procesamiento. Este pre-procesamiento alivia además la carga en el servidor, reduciendo enormemente el número de eventos a procesar, limitándolo sólo a los de potencial utilidad para detectar usability smells.

- La definición de un catálogo de *usability smells de interacción de usuario*

En el Capítulo 5 se describe un catálogo que describe 16 usability smells cuya detección puede automatizarse. La detección de usability smells, basada en la captura de eventos de usabilidad, permite a desarrolladores de diferentes niveles de experiencia en usabilidad tener un reporte rápido de problemas de usabilidad con un alto nivel de abstracción. Al capturar eventos de usuarios reales, los problemas más populares serán los primeros en reportarse.

- Un catálogo de *usability refactorings*

Para resolver los *usability smells* presentados a lo largo del trabajo, se presenta también un catálogo de *usability refactorings* que en la mayoría de los casos pueden generarse automáticamente. Una vez generados, los refactorings se aplican del lado del cliente, con lo cual no hace falta modificar el código de la aplicación.

- Un algoritmo para detectar la similitud entre elementos pequeños de interfaz web

Para lograr unificar la detección de un usability smell sobre diferentes elementos afectados que son equivalentes, se generó un algoritmo capaz de hallar, por un lado, la aparición de un mismo elemento en distintos contextos a pesar de pequeñas diferencias de presentación. Por otro lado, puede detectar familias de elementos equivalentes.

- Una herramienta que automatiza toda la propuesta

Se presentó Kobold, una herramienta que ofrece “Usabilidad como Servicio”, basada en todas las ideas desarrolladas en este trabajo. La herramienta es de código abierto, y se

encuentra online para ser utilizada, y está pensada como un framework que permite su fácil extensión.

- Validaciones empíricas para todas las etapas del proceso

Cada etapa del proceso fue validada con experimentos, incluso el algoritmo de similitud. En la sección siguiente se presenta un resumen con todos los resultados principales.

8.2. Validaciones

Todas las etapas del trabajo presentado fueron validadas con experimentos. En una primera validación, previa a la implementación de Kobold, se comprobó que los refactorings de usabilidad web del lado del cliente pueden de hecho traer beneficios en los tres aspectos de la usabilidad según el estándar ISO/IEC 25010: efectividad, eficiencia y satisfacción en uso. Se halló que 11 de los 21 refactorings de usabilidad evaluados mostraron beneficios estadísticamente significativos en al menos uno de estos aspectos. Además, ningún refactoring resultó negativo (en términos estadísticos) en satisfacción.

Se realizaron también 3 validaciones sobre la herramienta, una para cada etapa del proceso. Sobre la captura de eventos de usabilidad, se realizó un análisis de precisión y cobertura (*precision & recall*), y se obtuvieron valores F2 altos para todas las heurísticas de captura que dependen de la interpretación automatizada, en promedio **0.7925** (desvío estándar **0.09392**). La validación sobre detección de usability smells, realizada en comparación con tests de usuario manuales, demostró también resultados positivos. La herramienta fue capaz de hallar **66.67%** del total de problemas encontrados, mientras que mediante el test manual se halló el **70.37%**, con una intersección del **37.04%**. Respecto a esta intersección, si sólo consideramos problemas hallados manualmente, Kobold encontró **52.63%**.

Finalmente, la validación sobre usability refactorings, si bien fue más acotada en número de participantes, también arrojó buenos resultados. Durante esta validación se comparó la usabilidad en uso de aplicaciones refactorizadas mediante la herramienta contra sus versiones originales, a través de tests de usuario. Se concluyó que las versiones refactorizadas siempre mostraron mejoras en los tres aspectos evaluados: efectividad, eficiencia y satisfacción (con la excepción de un solo resultado que no varió).

8.3. Limitaciones

Si bien el método propuesto provee asesoramiento y corrección de problemas de usabilidad prácticamente sin esfuerzo, esto implica algunas limitaciones.

Algunos problemas de usabilidad requieren razonamiento humano, por lo tanto, no pueden detectarse automáticamente con la implementación propuesta. Además, el número de usability smells que pueden detectarse está restringido a aquellos con los cuales se topan los usuarios, dado que todo está basado en los eventos de usabilidad. Este trabajo podría complementarse con análisis heurístico estático para ampliar el espectro de problemas detectables.

Otra limitante tiene que ver con la falta de datos de entrenamiento. Si bien esto es una decisión consciente para facilitar al máximo la instalación, en algunos casos podría ayudar a detectar problemas en tareas concretas. Existen trabajos que pueden hallar problemas analizando desvíos sobre comportamientos de referencia en tareas específicas, como por ejemplo un *checkout* de un carrito de compras o un registro.

La necesidad de proteger la privacidad de los usuarios se consideró como primordial, por eso los eventos se detectan de forma independiente, es decir, sin relación entre ellos que permita reconstruir sesiones de usuario, entre otras medidas. Una captura de sesiones completas, en cambio, permitiría potencialmente el análisis de nuevos usability smells que podrían ser incluso más significativos. En el futuro se podrían utilizar técnicas de anonimato (*anonymization*) para poder contar con esta posibilidad.

La instalación no requiere que el desarrollador indique absolutamente nada sobre el perfil de su aplicación. No obstante, podría considerarse la posibilidad de solicitar al desarrollador que indique el tipo de aplicación: con un mínimo de información, tal vez sería posible explotar más aún las heurísticas de detección. Por ejemplo, los usability smells que generalmente afectan un sitio web de un sitio de noticias no serán los mismos que afecten a un sitio de e-commerce.

Los usability refactorings automatizados también tienen limitaciones. Al ser implementados completamente del lado del cliente, no pueden ofrecer soluciones que requieran modificar el código de la aplicación. Además, si bien las pruebas han mostrado que son transparentes a los usuarios, podría suceder que una cantidad de refactorings grande signifique una carga de código o procesamiento que afecte visiblemente la

performance del sitio refactorizado. Se podrían utilizar técnicas de optimización para que, al generar el código de los refactorings, se almacene en archivos comprimidos hasta que deba cambiarse. Esto aceleraría la carga extra de código en el cliente, y aliviaría el trabajo del servidor en gran medida. Nuevamente, esto afecta a los refactorings generados automáticamente, aquellos que son sólo sugeridos dependerán de cómo los implemente el desarrollador.

8.4. Trabajo futuro

A partir del trabajo presentado, han surgido varias potenciales líneas de investigación. Algunas de ellas ya han comenzado a explorarse.

En primer lugar, la captura de usability events en el cliente, si bien está diseñada para no sobrecargar al servidor, podría llegar a ser problemática en sitios de alto tráfico. En estos casos, el gran volumen de eventos capturados no sería necesario, y bastaría con obtener una muestra significativa. Para conseguir esto se podrían obtener perfiles de usuario automáticamente a partir de cierta meta-información para segmentar la captura según el tipo de usuario, o para distribuirla con el fin de obtener siempre eventos significativos.

La herramienta implementada está pensada como un framework, lo que permite ser extendida con facilidad, tanto para definir nuevos usability smells como para agregar usability refactorings. Sin embargo, la creación de nuevos usability refactorings podría hacerse más sencilla definiendo un DSL (Lenguaje Específico de Dominio) que permita a los desarrolladores definir reglas basadas en los usability events presentes. Esto reforzaría la idea de hacer que la herramienta pueda ser utilizada por desarrolladores de diferentes niveles de formación en usabilidad.

Los usability refactorings, si bien ya han sido probados mediante la implementación de varios ejemplos, están aún en una etapa temprana de desarrollo, y es necesario investigar y resolver varios potenciales problemas. Uno de ellos es la aplicación de múltiples refactorings a la vez. Cuando esto sucede, podría pasar que dos o más usability refactorings interfieran entre sí, dado que podrían estar accediendo todos al mismo elemento DOM y modificándolos de maneras diferentes. Es necesario agregar medidas de seguridad para que esto no ocurra.

Por último, se comenzó a investigar sobre la evaluación de los usability refactorings. ¿Qué sucede luego de que se aplicó un refactoring en producción? Lo primero y más importante sería lograr medir si los aspectos de usabilidad han mejorado. Si bien en algunos casos es posible seguir analizando los eventos de usabilidad para comprobar que el usability smell original ya no está, esto no siempre tiene sentido, ya que en muchos casos la interfaz de usuario se altera al punto de que el problema original ya no puede capturarse. En cambio, se necesitan métricas estandarizadas para comprobar aspectos de usabilidad como eficiencia, efectividad y satisfacción (aunque para esto último tal vez haya que abandonar temporalmente la transparencia de la propuesta y solicitar la opinión del usuario final).

Una vez logrado el objetivo de medir usabilidad, se podría pensar en aplicar varios refactorings alternativos en paralelo para un mismo problema, ya que ciertos usability smells tienen múltiples maneras de resolverse. Esto sería similar a un ciclo de A/B testing, donde se ponen a prueba varias soluciones a la vez y se elige la que mejor funciona. De hecho, podrían generarse varias versiones del sitio en producción, donde en cada una conviven un conjunto diferente de refactorings. Si bien un versionado múltiple serviría para fines de testing de usabilidad, también se podría aprovechar para, por ejemplo, personalizar un sitio.

Por último, debería realizarse una validación sobre la adopción que podría tener una herramienta como Kobold para aplicaciones en producción. Una barrera para su adopción podría ser la confianza en una herramienta externa para modificar una aplicación que ya se encuentra corriendo. Aun así, Kobold podría funcionar como una herramienta experimental donde los desarrolladores puedan poner a prueba diferentes refactorings. En ese caso, la herramienta podría generar código para modificar la aplicación directamente, evitando un servicio intermedio e incrementando la confianza en el refactoring.

Referencias

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

Apaolaza, A., Harper, S. and 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. doi: 10.1145/2700171.2804453.

Apaolaza, A. and Vigo, M. (2017) 'WevQuery', *Proceedings of the ACM on Human-Computer Interaction*. ACM, 1(1), pp. 1–17. doi: 10.1145/3095806.

Atterer, R., Wnuk, M. and 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. doi: 10.1145/1135777.1135811.

Au, F. *et al.* (2008) 'Automated usability testing framework', *Proceedings of the ninth Australasian User Interface Conference (AUIC2008)*, 76(January), pp. 55–64. Available at: http://dl.acm.org/citation.cfm?id=1378349.

Aula, A., Khan, R. M. and Guan, Z. (2010) 'How does Search Behavior Change as Search Becomes More Difficult?', *CHI Exploratory Search*, pp. 35–44. doi: 10.1145/1753326.1753333.

Basili, V. R., Caldiera, G. and Rombach, H. D. (1994) 'The goal question metric approach', *Encyclopedia of Software Engineering*, 1, pp. 528–532. doi: 10.1.1.104.8626.

Bernaschina, C. *et al.* (2017) 'A Big Data Analysis Framework for Model-Based Web User Behavior Analytics', in. Springer, Cham, pp. 98–114. doi: 10.1007/978-3-319-60131-1_6.

Breslav, S., Khan, A. and 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. doi: 10.1145/2598153.2598168.

Brooke, J. (1996) 'SUS - A quick and dirty usability scale', *Usability evaluation in industry*, 189(194), pp. 4–7. doi: 10.1002/hbm.20701.

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

Buttler, D. (2004) 'A short survey of document structure similarity algorithms', *International Conference on Internet Computing*, pp. 3–9. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.125.1983&rep=rep1&type=pdf.

Cabot, J. and Gómez, C. (2008) 'A catalogue of refactorings for navigation models', in *Proceedings - 8th International Conference on Web Engineering, ICWE 2008*, pp. 75–85. doi: 10.1109/ICWE.2008.14.

Carta, T., Paternò, F. and 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. Available at: http://link.springer.com/chapter/10.1007/978-3-642-23768-3_29 (Accessed: 20 February 2014).

Cassino, R. *et al.* (2015) 'Empirical validation of an automatic usability evaluation method', *Journal*

of Visual Languages & Computing. Elsevier, 28, pp. 1–22. doi: 10.1016/j.jvlc.2014.12.002.

Castillo, J. C., Hartson, H. R. and Hix, D. (1998) 'Remote usability evaluation: can users report their own critical incidents?', in *CHI '98: CHI 98 conference summary on Human factors in computing systems*, pp. 253–254. doi: http://doi.acm.org/10.1145/286498.286736.

Chi, E. H. (2002) 'Improving Web Usability Through Visualization', *IEEE Internet Computing*, 6(2), pp. 64–71. doi: 10.1109/4236.991445.

Dan, O., Dmitriev, P. and White, R. W. (2012) 'Mining for Insights in the Search Engine Query Stream', *International Conference Companion on World Wide Web*, pp. 489–490. doi: 10.1145/2187980.2188091.

Dig, D. (2011) 'A refactoring approach to parallelism', *IEEE Software*, 28(1), pp. 17–22. doi: 10.1109/MS.2011.1.

Dingli, A. and Mifsud, J. (2011) 'Useful: A framework to mainstream web site usability through automated evaluation', *Journal of Human Computer Interaction (IJHCI)*. Available at: http://www.cscjournals.org/manuscript/Journals/IJHCI/Volume2/Issue1/IJHCI-19.pdf (Accessed: 10 September 2017).

Distante, D. *et al.* (2014) 'Business processes refactoring to improve usability in E-commerce applications', *Electronic Commerce Research*, 14(4), pp. 497–529. doi: 10.1007/s10660-014-9149-0.

van Duyne, D. K., Landay, J. a. and Hong, J. I. (2006) 'The Design of Sites'. Addison-Wesley.

Fernandez, A., Insfran, E. and Abrahão, S. (2011) 'Usability evaluation methods for the web: A systematic mapping study', *Information and Software Technology*, 53(8), pp. 789–817. doi: 10.1016/j.infsof.2011.02.007.

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

Gaber, M., Zaslavsky, A. and Krishnaswamy, S. (2005) 'Mining data streams: a review', *ACM Sigmod Record*. Available at: http://dl.acm.org/citation.cfm?id=1083789 (Accessed: 2 September 2017).

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. doi: 10.1007/s10209-013-0323-2.

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. and 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. and Distante, D. (2011) 'Refactoring for Usability in Web Applications', *IEEE Software*, 28(3), pp. 60–67. doi: 10.1109/MS.2010.114.

Genov, A. (2005) 'Iterative Usability Testing as Continuous Feedback: A Control Systems Perspective', *Journal of Usability Studies*, 1(1), pp. 18–27.

Gregg, D. G. and Walczak, S. (2010) 'The relationship between website quality, trust and price premiums at online auctions', *Electronic Commerce Research*, 10(1), pp. 1–25. doi: 10.1007/s10660-010-9044-2.

Grigalis, T. and Čenys, A. (2014) 'Using XPaths of inbound links to cluster template-Generated web pages', *Computer Science and Information Systems*, 11(1), pp. 111–131. doi: 10.2298/CSIS130416020G.

Grigera, J. *et al.* (2016) 'Assessing refactorings for usability in e-commerce applications', *Empirical Software Engineering*, 21(3), pp. 1224–1271. doi: 10.1007/s10664-015-9384-6.

Grigera, J. *et al.* (2017) 'Automatic detection of usability smells in web applications', *International Journal of Human-Computer Studies*, 97, pp. 129–148. doi: 10.1016/j.ijhcs.2016.09.009.

Grigera, J., Garrido, A. and Rossi, G. (2017) 'Kobold: Web Usability as a Service', *Automated Software Engineering - Tool Demonstrations*, p. to appear.

Harms, P. and 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.

Harrati, N. *et al.* (2016) 'Exploring user satisfaction for e-learning systems via usage-based metrics and system usability scale analysis', *Computers in Human Behavior*, 61, pp. 463–471. doi: 10.1016/j.chb.2016.03.051.

Hartson, H. R., Andre, T. S. and Williges, R. C. (2003) 'Criteria For Evaluating Usability Evaluation Methods', *International Journal of Human-Computer Interaction*. Lawrence Erlbaum Associates, Inc., 15(1), pp. 145–181. doi: 10.1207/S15327590IJHC1501_13.

Hilbert, D. M. and Redmiles, D. F. (2000) 'Extracting usability information from user interface events', *ACM Computing Surveys*, 32(4), pp. 384–421. doi: 10.1145/371578.371593.

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. doi: 10.1145/502115.502118.

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. doi: 10.1016/j.ijhcs.2005.06.002.

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. (2013) *Automated Web Site Evaluation: Researchers' and Practioners' Perspectives*. Available at: https://books.google.com.ar/books?hl=en&lr=&id=blL6BwAAQBAJ&oi=fnd&pg=PR13&dq=automated+website+evaluation+ivoty&ots=ZkKpXd4ebq&sig=xzefTygheFXYfgFejUSfnrTWpFw (Accessed: 10 September 2017).

Ivory, M. Y. and Hearst, M. a. (2001) 'The state of the art in automating usability evaluation of user interfaces', *ACM Computing Surveys*, 33(4), pp. 470–516. doi: 10.1145/503112.503114.

Joshi, S. *et al.* (2003) 'A Bag of Paths Model for Measuring Structural Similarity in Web Documents', *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 9, pp. 577–582. doi: 10.1145/956750.956822.

Jurca, G., Hellmann, T. D. and Maurer, F. (2014) 'Integrating Agile and User-Centered Design: A Systematic Mapping and Review of Evaluation and Validation Studies of Agile-UX', in *2014 Agile Conference*. IEEE, pp. 24–32. doi: 10.1109/AGILE.2014.17.

Juristo, N. and Moreno, A. (2010) *Basics of software engineering experimentation*. Springer Publishing Company, Incorporated. Available at: http://dl.acm.org/citation.cfm?id=1965114 (Accessed: 28 April 2014).

Krug, S. (2000) 'Don't make me think!: a common sense approach to Web usability'.

Lanza, M. and 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. doi: 10.1080/01449290410001669914.

Nakamichi, N. *et al.* (2006) 'Detecting low usability web pages using quantitative data of users' behavior', *Proceedings of the 28th international conference on Software engineering*, pp. 569–576. doi: 10.1145/1134285.1134365.

Nanard, M., Nanard, J. and Kahn, P. (1998) 'Pushing Reuse in Hypermedia Design: Golden Rules, Design Patterns and Constructive Templates', in *Proceedings of the ninth ACM conference on Hypertext and hypermedia : links, objects, time and space---structure in hypermedia systems links, objects, time and space---structure in hypermedia systems - HYPERTEXT '98*. New York, New York, USA: ACM Press, pp. 11–20. doi: 10.1145/276627.276629.

Nebeling, M., Speicher, M. and 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. doi: 10.1145/2480296.2480303.

Nebeling, M., Speicher, M. and 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. doi: 10.1145/2470654.2481319.

Nielsen, J. (1994) *Usability engineering*. Available at: https://books.google.com.ar/books?hl=en&lr=&id=DBOowF7LqIQC&oi=fnd&pg=PP1&dq=Usability+Engineering+nielsen&ots=Bl06URMPzV&sig=g5yL3leY-wMs1f_FGdX0ZDkAjyA (Accessed: 3 September 2017).

Nielsen, J. and Loranger, H. (2006) *Prioritizing Web Usability*. Pearson Education. Available at: http://books.google.com/books?id=YQsje6Ecl4UC&pgis=1 (Accessed: 5 November 2013).

Norman, D. A. (2013) 'The Design of Everyday Things', *Human Factors and Ergonomics in Manufacturing*, p. 272. doi: 10.1002/hfm.20127.

Okada, H. and Fujioka, R. (2008) 'Automated Methods for Webpage Usability & Accessibility Evaluations', in Pinder, S. (ed.) *Advances in Human Computer Interaction*. InTech, pp. 351–364. doi: 10.5772/81.

Omer, B., Ruth, B. and Shahar, G. (2012) 'A New Frequent Similar Tree Algorithm Motivated by DOM Mining Using RTDM and its new variant – SiSTeR'.

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

Panach, J. I. *et al.* (2015) 'A framework to identify primitives that represent usability within Model-Driven Development methods', *Information and Software Technology*, 58, pp. 338–354. doi: 10.1016/j.infsof.2014.07.002.

Paternò, F., Schiavone, A. G. and Pitardi, P. (2016) 'Timelines for Mobile Web Usability Evaluation', in *Proc. Int. Working Conference on Advanced Visual Interfaces - AVI '16*. New York, New York, USA: ACM Press, pp. 88–91. doi: 10.1145/2909132.2909272.

Reis, D. C. *et al.* (2004) 'Automatic web news extraction using tree edit distance', *Proc. 13th WWW Conference*, p. 502. doi: 10.1145/988672.988740.

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

Saadawi, G. M. *et al.* (2005) 'A Method for Automated Detection of Usability Problems from Client User Interface Events AMIA 2005 Symposium Proceedings Page - 654 AMIA 2005 Symposium Proceedings Page - 655', pp. 654–658.

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

Computer Studies. Elsevier, 76, pp. 40–49. doi: 10.1016/j.ijhcs.2014.12.005.

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. doi: 10.1145/2556288.2557265.

Seffah, A. *et al.* (2006) 'Usability measurement and metrics: A consolidated model', *Software Quality Journal*, 14(2), pp. 159–178. doi: 10.1007/s11219-006-7600-8.

Shah, I. (2008) 'Event Patterns as Indicators of Usability Problems', *Journal of King Saud University - Computer and Information Sciences*, 20, pp. 31–43. doi: 10.1016/S1319-1578(08)80003-1.

Silva da Silva, T. *et al.* (2011) 'User-Centered Design and Agile Methods: A Systematic Review', in *2011 AGILE Conference*. IEEE, pp. 77–86. doi: 10.1109/AGILE.2011.24.

Speicher, M., Both, A. and Gaedke, M. (2015) 'S.O.S.: Does Your Search Engine Results Page (SERP) Need Help?', in *Proc. ACM Conf. on Human Factors in Comp Systems - CHI '15*. New York: ACM Press, pp. 1005–1014. doi: 10.1145/2702123.2702568.

Speicher, M., Both, A. and Gaedke, M. (no date) 'Was that Webpage Pleasant to Use ? Predicting Usability Quantitatively from Interactions'.

Torrente, M. C. S. *et al.* (2013) 'Sirius: A heuristic-based framework for measuring web usability adapted to the type of website', *Journal of Systems and Software*, 86(3), pp. 649–663. doi: 10.1016/j.jss.2012.10.049.

Vigo, M. and Harper, S. (2017) 'Real-time detection of navigation problems on the World "Wild" Web', *International Journal of Human Computer Studies*, 101, pp. 1–9. doi: 10.1016/j.ijhcs.2016.12.002.

Williams, L. and Cockburn, A. (2003) 'Agile software development: it's about feedback and change', *Computer*. IEEE Computer Society Press, 36(6), pp. 39–43. doi: 10.1109/MC.2003.1204373.

Wohlin, C. *et al.* (2012) *Experimentation in software engineering: an introduction*. Springer. Available at: http://books.google.com/books?hl=en&lr=&id=QPVsM1_U8nkC&oi=fnd&pg=PR5&dq=Experimentation+in+software+engineering:+an+introduction&ots=GNt7ofhPAw&sig=ZCo7HvId_SmPqH9cH-ifLYry13k (Accessed: 4 August 2014).

Wroblewski, L. (2008) 'Web Form Design: Filling in the Blanks', *Interactions*, 0(October), p. 226. doi: 10.1145/1374489.1374506.

Ying, M. and Miller, J. (2013) 'Refactoring legacy AJAX applications to improve the efficiency of the data exchange component', *Journal of Systems and Software*, 86(1), pp. 72–88. Available at: http://www.sciencedirect.com/science/article/pii/S0164121212002129 (Accessed: 28 April 2014).

Zanotti, M. (2016) *Accessibility and Crowdsourcing: Use of semantic tags to improve web application accessibility (in Spanish)*. Univ. of La Plata, Argentina.