Automatización de pruebas de compatibilidad web en un entorno de desarrollo continuo de software

Maximiliano Agustin Mascheroni, Mariela Katherina Cogliolo, Emanuel Irrazabal

Universidad Nacional Del Nordeste. Facultad de Ciencias Exactas y Naturales y Agrimensura. Departamento de Informática

{mascheroni, mcogliolo, eirrazabal}@exa.unne.edu.ar

Abstract. El desarrollo de software continuo, es un enfoque en el cual los equipos mantienen la producción de software en ciclos cortos de tiempo, asegurando que el producto pueda ser lanzado de manera fiable en cualquier momento. Hoy en día, este enfoque está siendo cada vez más utilizado en las organizaciones, especialmente en las que desarrollan aplicaciones en entorno web o móvil. Sin embargo, al lanzar versiones del producto con mayor frecuencia, emergen más defectos en el mismo. Esto se debe, principalmente, a que el tiempo para realizar los ciclos de pruebas son muy cortos. Uno de los desafíos que existe actualmente es la aceleración de las pruebas sobre la interfaz de usuario, entre ellas las de compatibilidad web. En este trabajo se presenta una técnica utilizada en una gran empresa de desarrollo de software, para automatizar las pruebas de compatibilidad de web, mediante la automatización de comparación de imágenes al hacer pruebas cruzadas entre distintos navegadores. Los resultados indican que la técnica propuesta se adapta a los requerimientos de los procesos de desarrollo continuo, aumentando el rendimiento y velocidad de este tipo de pruebas.

Keywords: pruebas de compatibilidad, comparación de imágenes, desarrollo continuo de software, pruebas automatizadas

1 Introducción

El Desarrollo de Software Continuo (DSC), es un enfoque de la Ingeniería de Software, en el cual, los equipos producen software en ciclos muy cortos de tiempo, asegurando el lanzamiento de diferentes versiones del producto en cualquier momento [1]. Dentro de este campo surgen diferentes técnicas y metodologías, por ejemplo: integración continua, despliegue continuo, entrega continua y pruebas continuas.

El DSC es un factor clave en las organizaciones como elemento diferenciador de la competencia y que además permite alcanzar mejoras rápida y eficientemente en las entregas a los clientes, brindándoles a los mismos un producto fiable [2]. Sin embargo, uno de los principales desafíos es mantener elevada la calidad del producto software. Al realizarse los despliegues del sistema con mayor frecuencia, aparecen

más defectos en el producto [3], [4]. Esto se debe, principalmente, a que el tiempo para realizar los ciclos de prueba son muy cortos.

Como una solución a este problema se ha intentado mantener la calidad del producto software utilizando las pruebas automatizadas. Mediante las mismas, se busca acortar los tiempos de entrega de los productos [5]. Así, por ejemplo, Google genera lotes de pruebas automatizadas utilizando herramientas especializadas [6], adquiriendo una mayor cobertura de pruebas antes de liberar una nueva versión del producto. Por el contrario, Mike Cohn y Anand Bagmar, ubican a las pruebas unitarias y a las de aceptación como base en las pirámides de pruebas [7] [8]; y las pruebas de interfaz gráfica (UI) se encuentran en el último lugar (ver **Fig. 1**).

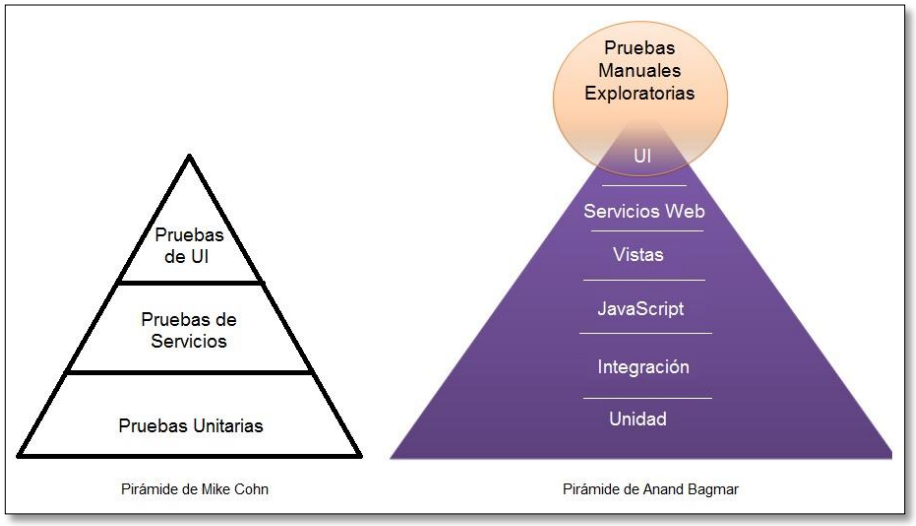

Fig. 1. Pirámides de Pruebas

Esto lleva a que muchas organizaciones pongan su mayor esfuerzo en automatizar pruebas unitarias y espacialmente, pruebas de aceptación. Así, por ejemplo, puede verse que los pequeños equipos de desarrollo software realizan principalmente pruebas de aceptación manual en el nordeste argentino [9].

Sin embargo, para otras empresas, es vital que el producto en desarrollo sea compatible en los distintos navegadores tanto en un computador, como en un dispositivo móvil. Las pruebas que se utilizan para determinar la compatibilidad de una aplicación en diferentes navegadores se denominan “pruebas de compatibilidad web” o “pruebas cruzadas entre navegadores” (más conocido en inglés como Cross-Browser Testing). En este trabajo se propone una técnica para automatizar las pruebas de compatibilidad web basándose en la comparación de imágenes mediante el procesamiento digital de imágenes.

Además de esta sección introductoria, el trabajo está dividido en 5 secciones. En la sección 2 se brinda un breve marco teórico sobre las pruebas de compatibilidad web. En la sección 3 se presenta la técnica propuesta y los algoritmos utilizados. Los

resultados de la implementación de la técnica se muestran en la sección 4. Finalmente, en la sección 5 se perfilan las conclusiones y trabajos futuro.

2 Pruebas de Compatibilidad Web

Que un sitio web sea compatible con todos los navegadores significa que se vea igual (o muy similar) en todos ellos. Algunos autores consideran como suficiente si el sitio puede ser percibido por el usuario con las mismas características, en los navegadores más importantes, como Internet Explorer, Firefox, Chrome, Opera, Safari y Mozilla [10].

El problema radica en que no todos los navegadores interpretan el código HTML y las hojas de estilo (CSS) de la misma manera [11]. Algunas de esas diferencias son tan importantes que provocan el mal funcionamiento del sitio o la perdida de visualización. Uno de los objetivos de la construcción de un sitio web es que pueda ser visitado por el mayor número de personas (y que éstas lo vean correctamente). Por ende, es muy importante que el sitio funcione igual en el mayor número de navegadores posibles [10]. Las Pruebas de Compatibilidad o Pruebas Cruzadas [12], son pruebas que se realizan en una aplicación determinada comprobando su compatibilidad con todos los navegadores de Internet y sistemas operativos del mercado. Para ello, existen diferentes técnicas:

- Verificación del cumplimiento de estándares (como, por ejemplo, W3C o ECMA): Consiste en analizar los componentes gráficos del sitio web en diferentes navegadores, para verificar que sigan los lineamientos y especificaciones de los estándares. Esta técnica se utiliza preferentemente en etapas intermedias del proceso de desarrollo, teniéndose que el diseño real se estará llevando a cabo con la consideración de los estándares que correspondan [13], [14].
- Pruebas de interfaz de usuario (UI): Es la más común de las técnicas. Puede ser realizada de forma manual o con software especializado (pruebas automatizadas). El objetivo de este tipo de prueba es revisar el contenido visual del sitio Web a través de la navegación de sus páginas en los diferentes navegadores [15], [16].
- Análisis del modelo de objetos del documento (DOM): Es una técnica dinámica que consiste en comparar el comportamiento de una aplicación web en diferentes navegadores, identificando las diferencias como defectos. La comparación se realiza combinando analisis estrucural de la información en el DOM junto con un análisis visual de la página en cuestión. [17].
- Comparación de Imágenes: Ésta técnica se basa en tomar una captura de pantalla del sitio en un tipo de navegador, y compararla con otra captura del sitio en otro navegador diferente del primero. Si ambas imágenes coinciden, entonces el sitio será compatible entre ambos navegadores [18].

De todas ellas, la técnica de comparación de imágenes es elegida por diversos autores [18], [19], [20], [21], [22] como la más adecuada para realizar pruebas de compatibilidad web, debido a la facilidad para ponerla en marcha.

3 Técnica Propuesta

En un ambiente de desarrollo continuo, las pruebas funcionales automatizadas cumplen un papel fundamental, ya que permiten realizar regresiones de manera rápida en pequeños ciclos de desarrollo por cada cambio introducido en el producto software. Con la aparición de los servidores de integración continua, es posible detectar la introducción de defectos en el código fuente lo antes posible [23]. De todas maneras, los problemas de incompatibilidad son detectados por equipos encargados de realizar pruebas manuales exploratorias sobre los diferentes navegadores. Estos equipos muchas veces se apoyan en software especializado para realizar comparación de imágenes.

En la actualidad, existe gran cantidad de herramientas no propietarias que permiten implementar algoritmos de comparación de imágenes [24]. Sin embargo, el problema de estas herramientas es que ninguna puede ser acoplada a la ejecución de pruebas funcionales sobre los sitios web [19].

La técnica propuesta consiste, por tanto, en complementar el proceso de pruebas funcionales sobre un sitio web, con un algoritmo de comparación de imágenes automatizado, y de esta manera acelerar la detección de incompatibilidades entre los diferentes navegadores.

El entorno fue desarrollado utilizando Java y consta de los siguientes componentes:

- Una herramienta para interactuar con los diferentes navegadores.
- Una herramienta para verificar el cumplimiento de condiciones (existencia de elementos, comportamientos esperados, etc.)
- Una herramienta para procesar imágenes.
- Un algoritmo para realizar la comparación de imágenes.
- Un mecanismo para ejecutar pruebas en paralelo sobre diferentes sistemas operativos y navegadores.
- Una herramienta de generación de reportes para mostrar los resultados de la ejecución de las pruebas.

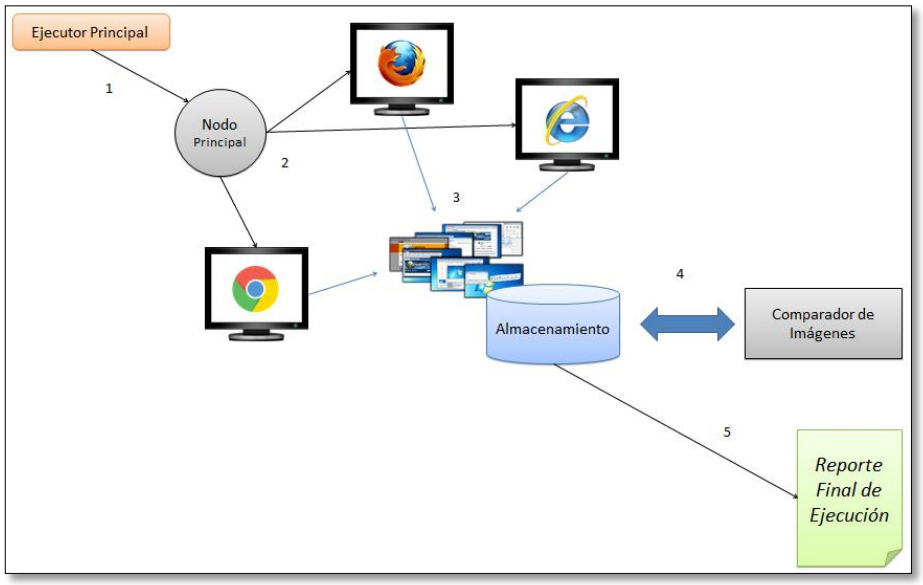

Fig. 2. Flujo de Ejecución de las Pruebas

El flujo de ejecución puede verse en la **Fig. 2**:

1. Las pruebas son ejecutadas a través de Maven. Esta herramienta se encarga de compilar el código y darle la orden al nodo principal para que comience su ejecución.
2. El nodo principal es un concentrador de Selenium Grid, un servidor que permite disparar hilos de ejecución de pruebas sobre múltiples nodos en paralelo. Este servidor instancia los navegadores configurados en cada nodo. Las pruebas son generadas utilizando Selenium Webdriver.
3. Durante la ejecución de las pruebas, se realizan capturas de pantalla y se almacenan en un disco duro al que tienen acceso todos los nodos de manera sincronizada. Las pruebas concluyen cuando todas las validaciones funcionales son completadas. Estas validaciones se realizan utilizando TestNG.
4. El algoritmo de comparación de imágenes toma las capturas de pantallas generadas, y comienza a analizarlas en pares generando otras imágenes *resultados*, que luego las almacena en el mismo disco donde se encuentran las capturas.
5. Cuando finaliza el proceso, se genera el reporte final con los resultados de la ejecución de las pruebas en un archivo HTML.

3.1 El proceso de captura de pantallas

La interacción con el sitio web es responsabilidad de la herramienta Selenium Webdriver. Un mismo script de prueba es ejecutado en distintos navegadores en paralelo, mediante una configuración de Selenium Grid que permite instanciar un hilo

de ejecución por cada máquina virtual (nodo) del entorno. A medida que estas pruebas van avanzando de una página del sitio a otra, se va tomando una captura de pantalla.

Una de los desafíos que se presentó fue buscar el mecanismo más adecuado para obtener diferentes capturas de pantalla con las mismas dimensiones. Una alternativa fue utilizar una función dada por la herramienta Selenium Webdriver para obtener capturas de pantallas del sitio. Pero esto generó problemas con las capturas del navegador Mozilla Firefox, donde las mismas no solo contenían lo que se vería en la pantalla, sino todo el contenido de la página a la cual se accede realizando el desplazamiento vertical (scrolling-down/up). La segunda opción fue utilizar una API de Java, que permite tomar capturas de pantalla del computador. Sin embargo, si bien los resultados fueron imágenes con las mismas dimensiones, todas ellas contenían la barra de herramientas de cada navegador, y al momento de realizar la comparación de imágenes las pruebas de compatibilidad siempre fallaban.

Finalmente, se optó por un algoritmo que consiste en tres pasos. El código fuente escrito en lenguaje Java puede verse en la **Fig. 3**.

1. Colocar el navegador en modo “pantalla completa”.
2. Obtener la dimensión total de la pantalla utilizando la API Toolkit de Java.
3. Generar la captura de pantalla a través de la API Robot de Java.

```
//Paso 1
Actions actions = new Actions(navegador);
actions.sendKeys(Keys.F11).perform();

//Paso 2
Dimension dimension = Toolkit.getDefaultToolkit().getScreenSize();
Rectangle rectangle = new Rectangle(dimension);

//Paso 3
BufferedImage image = new Robot().createScreenCapture(rectangle);
ImageIO.write(image, "png", new File(path));
```

Fig. 3. Método para tomar captura de pantalla de un nodo con Windows.

3.2 Algoritmo de Comparación de Imágenes utilizado

Las imágenes tomadas son almacenadas en el disco duro de un computador, con una nomenclatura que representa: el identificador de la prueba; el nombre de la página en la que se tomó la captura; el navegador; y el nombre y versión del sistema operativo del nodo. Por ejemplo: Test1802-detalle_itinerario-chrome-win8.1.png; Test706-pagina_pago-ff_v44-ubuntu12.2.png.

La herramienta para realizar las verificaciones de los resultados esperados y condiciones (TestNG), presenta un mecanismo para ejecutar instrucciones al concluir las pruebas. Este mecanismo se define mediante las anotaciones de @AfterSuite/@AfterTest/@AfterClass. Asímismo, permite la ejecución de

instrucciones previa a la de las pruebas (utilizando los recíprocos @BeforeSuite/@BeforeTest/@BeforeClass).

Para realizar la comparación de imágenes se utiliza la anotación @AfterSuite. Dentro del método que contiene esta anotación, se realiza la llamada al algoritmo de comparación de imágenes.

El primer paso en esta etapa consiste en obtener todas las imágenes correspondientes a un determinado nodo, utilizando el navegador y sistema operativo especificados en el nombre de las imágenes. Luego, se obtienen las imágenes correspondientes a un segundo nodo. Una vez obtenidas las imágenes correspondientes a los dos primeros nodos, se utiliza un algoritmo que toma una imagen de un nodo y busca su correspondiente (según el identificador y nombre de página) entre las capturas del otro nodo (ver **Fig. 5**). La comparación de imágenes comienza a realizarse de a pares.

```
List<BufferedImage> imagenesComp0 = getImagesMatchingWith("chrome45-win7");

//Primer Comparacion
List<BufferedImage> imagenesComp1 = getImagesMatchingWith("ff_v44-ubuntu12.2");
startComparison(imagenesComp0, imagenesComp1);

//Segunda Comparacion
List<BufferedImage> imagenesComp2 = getImagesMatchingWith("ff_v44-ubuntu12.2");
startComparison(imagenesComp0, imagenesComp2);

//Ultima Comparacion
startComparison(imagenesComp1, imagenesComp2);
```

Fig. 4. Proceso de toma de todas las imágenes de los diferentes nodos

```
private void startComparison(List<BufferedImage> base, List<BufferedImage> comp) {
    for (BufferedImage image: base) {

        //Se obtiene datos de la primer imagen de la lista
        String imgBase = Utils.getIDAndPagName(image);

        //Se busca la imagen correspondiente en la segunda lista
        BufferedImage corresp: Utils.getImageMatchingWith(imgBase);

        //Se realiza la comparación de ambas imágenes
        Compare.startComparison(image, corresp);

    }
}
```

Fig. 5. Obtención de pares de imágenes para ser comparadas

Finalmente, el algoritmo de comparación de imágenes se ejecuta:

1. Se verifica que ambas imágenes tengan las mismas dimensiones. Si no son iguales, se ignora la comparación y se obtiene un fallo.

2. Se obtiene una matriz de pixeles de la imagen base, y se comienza a recorrer todos los pixeles uno por uno, comparándolos con su recíproco en la otra imagen. En caso de ser diferentes, se guarda la posición de los pixeles diferentes.
3. Si no hubo diferencias, la prueba de compatibilidad ha sido completada con éxito. En caso de fallo, se genera una tercera imagen resultado de la comparación. La imagen es pintada con puntos en cada posición correspondiente a los pixeles diferentes. Esta imagen es guardada con un formato de mapa de calor junto al par de capturas que fueron analizadas.

```
private boolean bufferedImagesEqual(BufferedImage img1, BufferedImage img2, String path)
    throws IOException {

        boolean isEqual;

        if (img1.getWidth() == img2.getWidth() && img1.getHeight() == img2.getHeight()) {
            BufferedImage result = new BufferedImage(img1.getWidth(), img1.getHeight(),
                BufferedImage.TYPE_INT_RGB);

            for (int x = 0; x < img1.getWidth(); x++) {
                for (int y = 0; y < img1.getHeight(); y++) {
                    if (img1.getRGB(x, y) != img2.getRGB(x, y)) {
                        isEqual = false;
                        result.setRGB(x, y, Color.ORANGE.getRGB());
                    }
                }
            }

            ImageIO.write(result, "png", new File(path));

        } else {
            isEqual = false;
        }

        isEqual = true;

        return isEqual;
    }
```

Fig. 6. Algoritmo de comparación de imágenes

Por último, el resultado de las comparaciones puede ser observado en el reporte final. El reporte final es generado con ReportNG, como un complemento de TestNG. Para la sección de pruebas de compatibilidad se ha desarrollado un reporte HTML que tiene el formato mostrado en la **Fig. 7** al cual se accede haciendo click en cada fallo de la comparación.

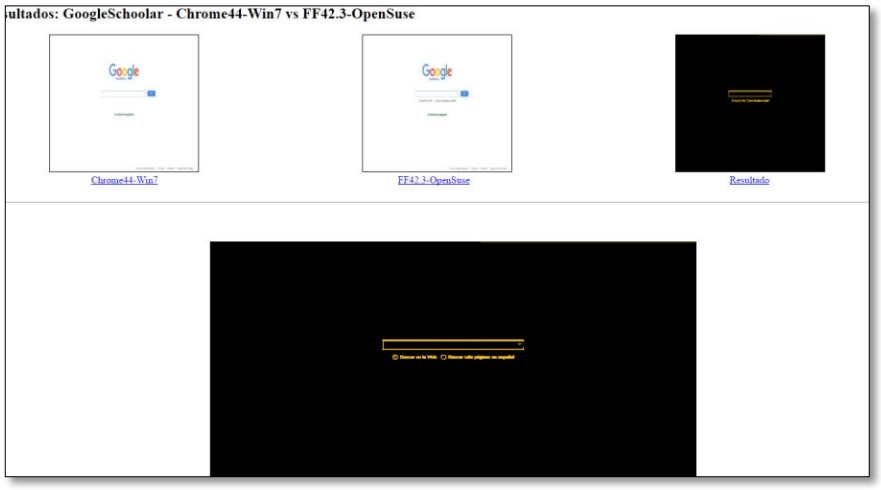

Fig. 7. Ejemplo de un reporte detallado de una comparación.

4 Resultados obtenidos

Esta técnica ha sido implementada en un ambiente real de desarrollo continuo, y los resultados demuestran su eficiencia. Por un lado, la implementación del algoritmo fue una tarea muy sencilla de llevar a cabo por el equipo de desarrollo. La técnica ha acelerado un 92% el tiempo de ejecución de las pruebas de compatibilidad y por ende toda la etapa de pruebas (ver **Fig. 8**). Una prueba de compatibilidad convencional realizada por un miembro del equipo lleva aproximadamente 10 minutos dependiendo de la página. Una prueba de compatibilidad en la misma página con la técnica propuesta lleva aproximadamente 1 minuto, requerido para observar el mapa de calor con los resultados. En la **Fig. 9**, puede observarse una disminución en el tiempo total del proceso de liberación de cada versión del sitio, a partir de la implementación de la técnica propuesta.

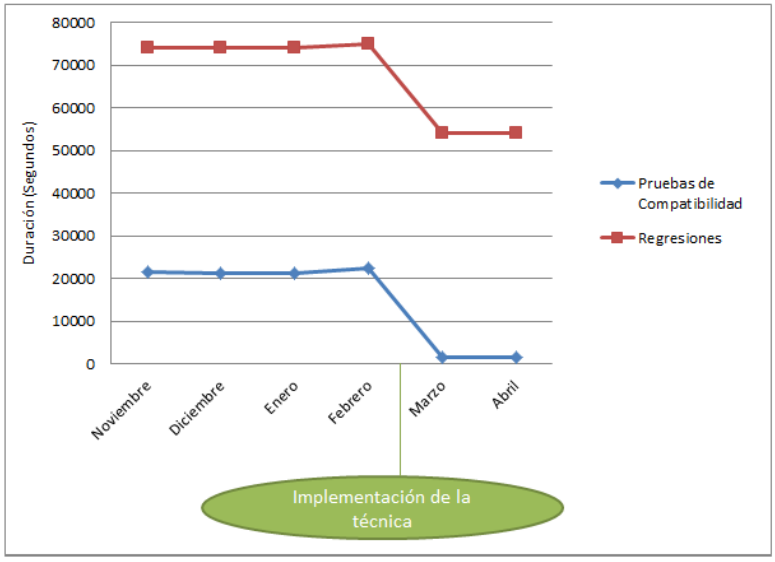

Fig. 8. Tiempo de ejecución de las pruebas en los últimos 6 meses (en segundos)

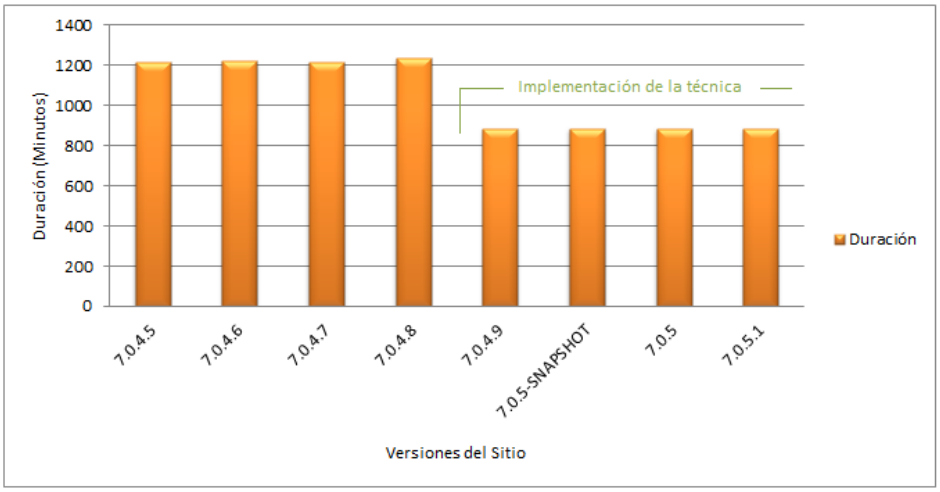

Fig. 9. Tiempo total del proceso de liberación de las últimas 8 versiones del producto (en minutos)

Sin embargo, también se realizaron sugerencias para evitar falsos fallos en las comparaciones de páginas que contienen anuncios publicitarios cambiantes o ventanas emergentes localizadas en diferentes secciones.

5 Conslusión y Trabajo Futuro

La técnica propuesta es una iniciativa para automatizar el proceso de pruebas de compatibilidad mediante un algoritmo de comparación de imágenes. Para validarla, ha sido implementada en una gran empresa que trabaja con desarrollo continuo de software. Los resultados demuestran que la herramienta permite acelerar el proceso de pruebas, mediante la automatización de pruebas de compatibilidad web.

Por un lado, los tiempos de ejecución de pruebas disminuyeron un 82%, y esto también redujo el tiempo total del proceso de liberación de las versiones del sitio. Además permite generar reportes en formato HTML muy fáciles de comprender.

Con la implementación de este enfoque, se eliminan las exhaustivas pruebas de visualización de componentes en busca de incompatibilidades sobre los distintos navegadores. De esta manera, las tareas manuales quedan reducidas simplemente a observar los reportes. Asímismo, permite detectar más defectos que pueden ser pasados por alto cuando se realizan las pruebas manualmente.

Sin embargo, se encontraron inconvenientes en la comparación de sitios que contienen diferentes publicidades y/o elementos emergentes, produciendo falsos fallos en las pruebas.

Como trabajo futuro se propone, en primer lugar, mejorar el algoritmo implementado a modo de evitar comparaciones en elementos no pertenecientes al sitio web (como, por ejemplo anuncios). En segundo lugar, se harán uso de herramientas más actuales de virtualización (como, por ejemplo la herramienta Docker) para poner en ejecución las pruebas funcionales.

Finalmente, se investigarán técnicas que puedan ser implementadas para evitar el trabajo manual que se requiere para observar los resultados de las comparaciones, de manera de hacer el proceso totalmente automatizado. También se buscará ajustar el algoritmo para evitar escenarios alarmantes a causa de pequeñas diferencias entre los navegadores.

Agradecimientos

Este trabajo se ha realizado en el marco del proyecto PI-F10-2013 “Métodos y herramientas para la calidad del software”, acreditado por la Secretaría de Ciencia y Técnica de la Universidad Nacional del Nordeste (UNNE) para el periodo 2014-2017.

Referencias

1. B. Fitzgerald and K. J. & Stol, "Continuous software engineering and beyond: trends and challenges," in Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering, 2014, pp. 1-9.
2. L. Chen, "Continuous delivery: Huge benefits, but challenges too.," Software, IEEE, vol. 32, no. 2, pp. 50-54, 2015.

3. G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way.," Information and Software technology, vol. 57, pp. 21-31, 2015.
4. H. H. Olsson and J. Bosch, "Towards agile and beyond: an empirical account on the challenges involved when advancing software development practices," in Agile Processes in Software Engineering and Extreme Programming. Roma, Italia: Springer International Publishing, 2014, pp. 327-335.
5. E. Dustin, T. Garrett, and B. Gauf, Implementing automated software testing: How to save time and lower costs while raising quality.: Pearson Education, 2009.
6. S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 2014, pp. 235-245.
7. M. Cohn, Succeeding with agile: software development using Scrum.: Pearson Education, 2010.
8. A. Bagmar, "Enabling Continuous Delivery in Enterprises with Testing," in Agile2015, India, 2015.
9. G. N. Dapozo et al., "Características del desarrollo de software en la ciudad de Corrientes," in XXI Congreso Argentino de Ciencias de la Computación, Junin, 2015.
10. J. Liberty and D. Maharry, Programming ASP.NET 3.5, 4th ed. Massachusetts, USA: O'Reilly Media, 2008.
11. J. R. Aranda Córdoba, Desarrollo y reutilización de componentes software y multimedia mediante lenguajes de guión. Málaga, España: IC Editorial, 2014.
12. J. Aracil. (2015) Globe Testing. [Online]. http://www.globetesting.com/2012/07/pruebas-de-compatibilidad/
13. W3C. (2003) W3C España. [Online]. http://www.w3c.es/estandares/
14. Ecma Standards. (2013) Ecma International. [Online]. http://www.ecma-international.org/publications/standards/Standard.htm
15. A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in Proceedings of The 10th Working, Florida, USA, Nov, 2003.
16. Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI-based software applications," ACM Transactions on Software Engineering and Methodology, vol. 16, no. 1, p. 4, Febrero 2007.
17. S. R. Choudhary, Versee H., and A. Orso, "WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications," in Software Maintenance (ICSM), Timișoara, Romania, 2010.
18. A. Hori, S. Takada, H. Tanno, and M. Oinuma, "An Oracle based on Image Comparison for Regression Testing of Web Applications," in International Conference on Software Engineering and Knowledge Engineering (SEKE), Pittsburgh, USA, 2015.
19. E Selay, Z. Q. Zhou, and J Zou, "Adaptive random testing for image comparison in regression web testing," in Digital lmage Computing: Techniques and Applications (DlCTA). International Conference. IEEE, 2014, pp. 1-7.
20. T. Saar, M. Dumas, M. Kaljuve, and N. Semenenko, Browserbite: cross-browser testing via image processing, Abril 2015, Software: Practice and Experience.
21. S. Mahajan and W. G. Halfond, "Finding HTML presentation failures using image comparison techniques," in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 91-96.

22. T. Saar, M. Dumas, M. Kaljuve, and N Semenenko, "Cross-browser testing in browserbite," Web Engineering. Springer International Publishing, no. 503-506, 2014.
23. P. M. Duvall, S. Matyas, and A. Glover, Continuous integration: improving software quality and reducing risk.: Pearson Education, 2007.
24. G. Smith. (2014, Febrero) Mashable. [Online]. http://mashable.com/2014/02/26/browser-testing-tools/#ZenWlmiAYGqa

25. C. Eaton and A. M. Memon, "An empirical approach to testing web applications across diverse client platform configurations," International Journal on Web Engineering and Technology (IJWET), vol. 3, no. 3, pp. 227-253, 2007.