Automatic Deformations Detection in Internet Interfaces: ADDII

Leandro Sanchez[2] and Plinio Thomaz Aquino Jr.[1,2(✉)]

[1] Centro Universitário da FEI – Fundação Educacional Inaciana
Pe. Sabóia de Medeiros, São Bernardo do Campo, São Paulo, Brazil
plinio.aquino@fei.edu.br
[2] IPT - Instituto de Pesquisas Tecnológicas, São Paulo, Brazil
leandrosanchez@me.com

Abstract. Developers have been trying to create uniform and consistent web-pages in the different browsers available in the market. Known as Crossbrowser issue, it affects pages in different ways, on its functionalities and visually aspects and sometimes not related to the source code. Using screenshot and image comparison algorithms, this paper presents a technique for automated detection of visual deformations in web pages using a tool developed during the research called Automatic Deformations Detection in Internet Interfaces (ADDII).

Keywords: Business: interfaces in automated manufacturing · Business: visual analytics and business intelligence · Technology: intelligent and agent systems

1 Introduction

Since the beginning of the Internet and offer of different web browsers, has been a challenge to developers to keep your pages uniform and consistent between the different versions available, both in functional and visually. Much of the problem not related to the source code created by developers, but by the different implementations of browsers, which interpret and visually present the pages. About four browsers have over 70 % of market share and big companies like Microsoft are still investing on new products, as Project Spartan [1] recently announced. If we add in that count the currently available browsers on other platforms, such as smartphones and tablets, the number would be higher. Different tools have been developed and commercially available to support developers in testing your pages and applications. However, none of them is completely effective in automatically detecting visual deformations, usually leaving the checking and interpretation on developers mind. This paper presents a technique for automated detection of visual deformations in web pages using a tool developed during the research called Automatic Deformations Detection in Internet Interfaces (ADDII). The ADDII compares screenshots of pages generated in three different browsers and uses algorithms to compare type image Perceptual Hash to verify the similarity between them, indicating which browsers had a discrepancy.

In the next chapters, we will present more detailed information about the Cross browser issue, the visual algorithm's, the concept and process behind ADDII and our experiment results.

M. Kurosu (Ed.): Human-Computer Interaction, Part III, HCII 2015, LNCS 9171, pp. 43–53, 2015.
DOI: 10.1007/978-3-319-21006-3_5

2 Cross Browser Issue: Visual or Functional?

During the research, the compatibility issue divided into two types: Visual and Functional. The first type refers to the rendering of a page, in the interpretation of the code by web browsers, including HTML, Javascript and CSS content and present visual differences or errors that mischaracterize content. The image below illustrates the visual issue, which the center content is dislocated to right side, not following the header that is in the left position (Fig. 1).

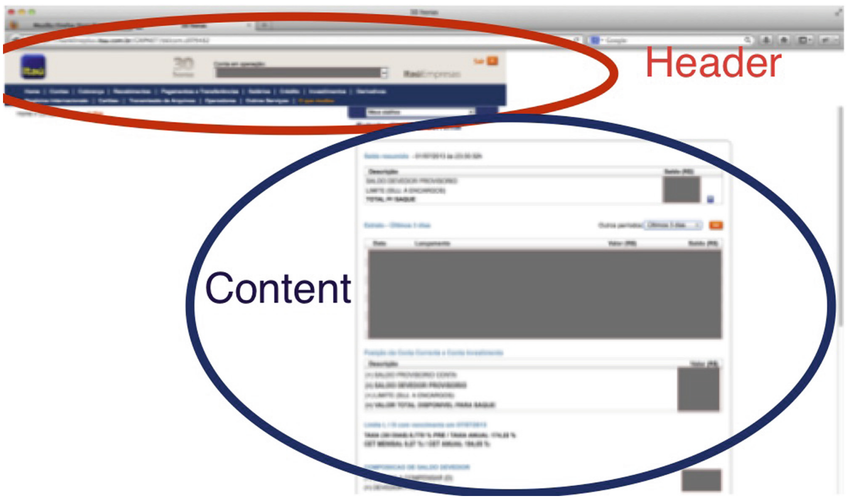


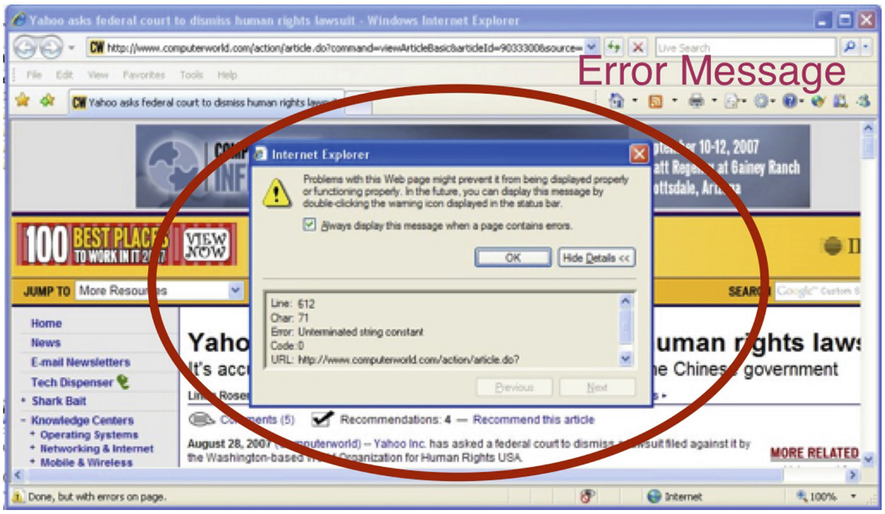
Fig. 1. Visual issue (source case: itaú internet banking, browser safari)

The second type comes to functional errors when the browser does not interpret a particular action or event correctly. Both types have common importance, however may contain different relevance depending on their use. The problem of type Visual directly affects developers who produce visual content, in which the pages created for advertising, promotion and marketing, where the inadequate presentation of a given page implies that a message not delivered to the user. The image below illustrate the functional issue, which the browser throws an error message during the execution in one of the functions (Fig. 2).

Develop applications compatible with most existing browsers is still a problem for developers, due to implementation differences in most of its components, which interpret and render the codes in different ways. Researchers Grosskurth et al. [2] presented the eight (8) components that make up the architecture of Internet browsers: User Interface; Browser Engine; Rendering Engine; Networking Subsystem; Javascript Interpreter; XML Parser Subsystem; Display Backend and Data Persistence Subsystem.

Also during the research, found that the companies that develop web browsers implement these components in different ways.

Figures 3 and 4 show two different browsers architecture. You can see the different components. The areas of User Interface, Browser Engine, Rendering Engine, Data Persistence, Networking, JavaScript Interpreter, XML (eXtensible Markup Language)


Yahoo asks federal court to dismiss human rights lawsuit - Windows Internet Explorer
Error Message
Internet Explorer
OK
Hide Details <<
Previous
Next
Done, but with errors on page.
Internet
100%

Problems with this Web page might prevent it from being displayed properly or functioning properly. In the future, you can display this message by double-clicking the warning icon displayed in the status bar.

Always display this message when a page contains errors.

```
Line:  612
Char:  71
Error: Unterminated string constant
Code: 0
URL:  http://www.computerworld.com/action/article.do?
```

Fig. 2. Functional issue (source: computerworld (2007), browser IE 6)

Parser and Display backend have differences in their implementations. The most noticeable to the user is the user interface (User Interface), which explicitly differs between browsers. Components such as networking, because they are different, expose specific security vulnerabilities of each browser and therefore rarely the same vulnerability found is contained in all browsers. Other components have a direct influence on performance, such as *JavaScript Interpreter*. Their different implementations make a browser more efficient for execution of specific scripts on pages. It is common to find JavaScript codes that do not work in all browsers. As an example, *document.getElementById* statement, which should return the unique identifier of an object on a page in all browsers, in some versions of Internet Explorer may return unexpected values as Microsoft article [3].

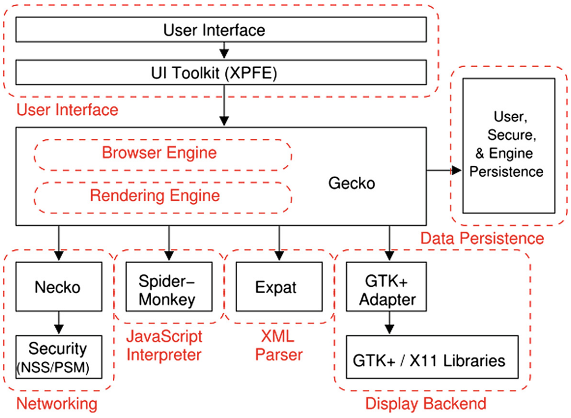


Fig. 3. Mozila architecture (source: [2] architecture and evolution of the modern web browser, p. 9).

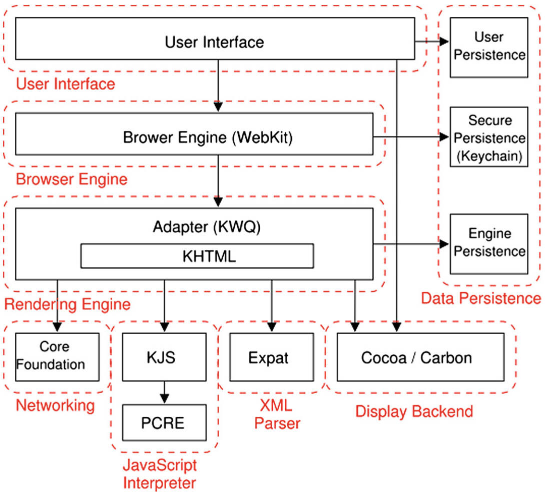


Fig. 4. Safari architecture (source: [2] architecture and evolution of the modern web browser, p. 14)

Studies continue to exist in this area, implementing different forms of check or normalize the pages, making it compatible with different browsers. The researchers Eaton et al. [4] demonstrated that the use of a tool that performs scanning in HTML (Hyper Text Markup Language) code to find invalid tags or used incorrectly, causing inconsistency in the application. Have the researchers Choudhary et al. [5] proposed the use of a tool that performs validations in the DOM (Document Object Model) structure of pages and in the analysis of images (screenshots). The researchers, Zhu et al. [6] demonstrated a technique for creating layouts, which contains a generator able to produce HTML code set for each version, and model browser.

3 Visual Comparison Algorithm's

During the research, this work tried to explore the use of image comparison algorithms, which can point the similarity between two images. Algorithms that address this issue are being developed in the areas of computer graphics and vision. Perhaps the most common algorithms are the Peak signal-to-noise ratio (PNSR) and root-mean-square error (RMSE), which are used to measure the quality of images. These algorithms can be found as library functions called *ImageMagick* [7]. An evolution of these algorithms is the Structural Similarity Image Metric (SSIM), presented by Wang et al. [8], which computes and compares the number of errors on the images to see their similarities. An implementation of this algorithm can be found in the Python-based tool called ***pyssim*** [9]. After validate these algorithms, the visual perception chosen, for its efficacy during the tests. The RMSE and PNSR algorithms were not effective to check the discrepancies and presented the results in decibels. The implementation of SSIM has values between 0 and 1, complicating the calculation to check similarity on images of different sizes. In the other hand, visual perception algorithms presented the results with numerical values that facilitate the calculation, including the number of different pixels between the two images.

There are several algorithms using Hash and its most common applications are file integrity checking. The MD5 and SHA1 algorithms have become popular on authentication systems, since the hash is unique and unchanging as a signature [10], reflecting the exact sequence of bytes. In case of any change in the bytes, the Hash will be different. These algorithms would be efficient if the scope is to validate if two images are identical, exactly same in the bytes point of view.

The Perceptual Hash allows you to check the similarity between two images, from small changes imperceptible to the human eye, including small changes of color and form.

Some researchers have been working on new algorithms and improving some existing ones over the years. Lin et al. [11] and Fridrich et al. [12] studies showed that verify the differences of two coefficients resulting from the calculation of the DCT (Discrete Cosine Transform), which is a formula used in processing digital and compression algorithms. Although they worked on different implementations, both used the DCT to calculate the Hash. Another algorithm is the SVD (Singular Value Decomposition). Kozat et al. [13] proposed an algorithm to calculate the hash using this formula. Perceptual Hash algorithms implementations are available for many different languages and platforms. For the development of ADDII, we used the algorithm written by Shepherd [14], which returns the Hamming Distance: given two strings, the Hamming distance is the lower number of replacements required to transform a string in another, or number of errors that turned on each other [15]. The Shepherd algorithm [14] was based on the article published by Krawetz [16].

The Perceptual Diff, created by Yee et al. [17] allows the comparison of two images, indicating whether they are similar or not and the number of different pixels. The comparison method used for calculating metrics of processed images, extending the VDP (Visible Differences Predictor) technique by Daly [18]. The comparison performed by the Perceptual Diff shows the differences in pixel by pixel in each image area available, taking into account the scanning angle. The algorithms and source code can be accessed through the SourceForge site ***pdiff*** [17].

4 ADDII: Automatic Deformations Detection in Internet Interfaces

The ADDII has four steps to perform the verification of pages. First, it loads the URL’s of the pages to scanned, which should be available on web servers. The second step called ***Screenshot Generator***, performs the screenshot of the images of each URL in three different browsers. The third step compares each screenshot of each URL generated in the above process, recording the result of the two algorithms implemented. Finally, the last step present the results, showing how similar is each screenshot between each other. As mentioned before, the ADDII implements two Visual comparison algorithms. Both have different metrics but with the same goal. Pointing out the similarity between two images.

4.1 Process and Components Detail

The following diagram illustrates the communication between the components created with ADDII.

Figure 5 illustrates the ADDII Architecture. The first lane shows the main process, the second the storage layer, and the interaction with the file system in the third. The lanes are also divided into three steps (columns), the first column generation screenshots, the second resizing images for the same resolution, and the third processing of the image comparison algorithms.

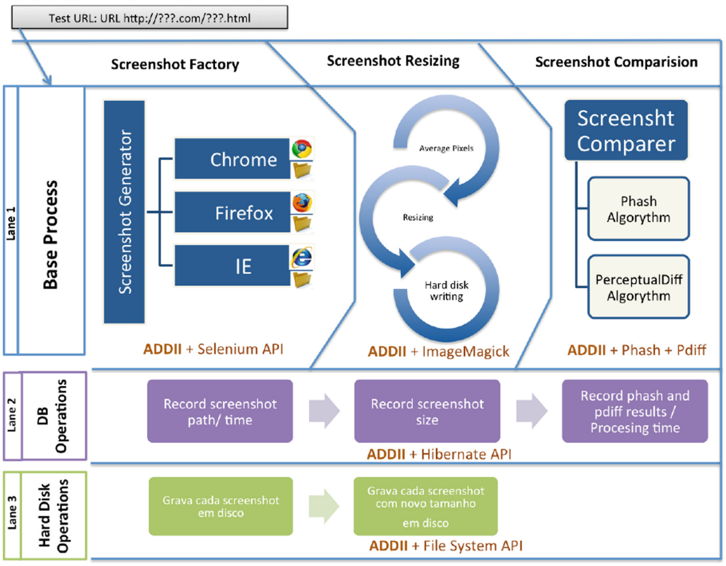


Fig. 5. Components and process (source: author)

When you run the ADDII, the first step is obtain the screenshots. This task leverage Selenium API to take screenshots. This same step stores the image in a folder in the file system and records the path in the database (lanes 2 and 3). The second step calculates and resizes the three screenshots obtained in the previous step. The third and final step, runs the two visual comparison algorithms implemented and collect the results recorded in the database.

The implementation of the above components demonstrates the modularity of the ADDII. The implementation of new browsers to obtain screenshots, through new future plugins offered by Selenium API or components developed apart is possible. New image comparison algorithms can also be added or improved in new versions.

4.2 Automatic Screenshots

To perform screenshots, ADDII utilize the Selenium Java API [19]. Selenium is an API that allows developers to perform dynamic actions on pages and sites across browsers.

It is widely used in Web Application testing automation. Selenium API requires that the browser being used is installed on the computer, the screenshots are generated directly in the browser itself. The code snippet below illustrates the Webdriver API call to perform the screenshot in Google Chrome browser.

```
public static void main ( String [ ] args ) {
/ / TODO Auto -generated method stub
  WebDriver driver = new ChromeDriver ( ) ;
  Dimension dim = new Dimension ( 1366, 768 ) ;
        driver.manage ( ) window ( ) setSize (dim ) . . ;
        driver.get ( args [ 0 ] ) ;
        try {

                File scrFile = ( ( TakesScreenshot )
driver ) getScreenshotAs ( OutputType.FILE ) . ;
                FileUtils.copyFile ( scrFile , new File (
args [ 1 ] ) ) ;
            } Catch ( IOException e1 ) {
                e1.printStackTrace ();
            }
        / / Close the browser
        driver.quit ();
```

[Code snippet from ADDII]

4.3 Visual Algorithm’s Implementation

The ADDII implements two algorithms visual algorithm’s. Both have different metrics but with the same goal. Pointing out the similarity between two images. The first algorithm implemented, was created by the programmer Elliot Shepherd [14], using Java language. In this work the algorithm is called pHash. The result of the algorithm is a range of values that informs how equal are two images. To perform the verification of images, it uses the DCT (Discrete Cosine Transformation) for low frequencies of the image, as used in image compressors like JPEG format. The Hamming distance algorithm is applied to calculate the difference between the Hash’s. When the algorithm returns 0–10, means that the images are similar. If returning more than 10 means that the images have significant differences.

The second algorithm is called Perceptual Image Diff [17], which the binaries are distributed as free software by the GNU (General Public License). Its use is performed through the command prompt, passing the path of the two images to be verified, as the example below:

```
Perceptualdiff image1.png image2.png
```

The algorithm returns a text saying that the images are similar. In verbose mode, activated via parameter *-verbose* also returns the number of distinct pixels was found between images. During the implementation of Perceptual Image Diff, was found that the utility only compares images with similar resolution. However, the screenshots generated by Selenium API had small differences in resolution. To resolve this point, the ADDII implemented a call to *ImageMagick API*, which among its many functions allows you to resize an image to a desired resolution. To configure the most appropriate resolution, we chose a simple calculation of average pixel width and height of the three screenshots generated. The formula is:

```
The formula is:
Img1.width + Img2.width + img3.width / 3 = Average width
of screenshots
Img1.height + Img2.height + img3.height / 3 = Average
height of screenshots
```

ImageMagick using the values calculated on the above formula converts the final resolution of the screenshots. On ADDII the two algorithms are complementary. The first shows that the images are similar, the second reports the amount of distinct pixels and how different two images are.

In the next chapter will present the test scenarios for validation ad results obtained with ADDII.

5 Results

In order to validate the effectiveness of ADDII, three (3) cases were prepared, containing visual errors in at least one of the browsers. In addition, ten (10) sites also checked from the Internet, just as validation of its operation outside of a controlled environment. The definition of right or wrong in the results table was defined *pHash algorithm*, the information of pixels being informed by the additional Perceptual Diff algorithm in the analysis. For each test was selected a default browser, which is the browser where the test or site in question works correctly. The default browser shown in bold in the table below, which shows the test results.

As shown in Table 1, cases 1 and 3 showed similar results where the algorithm pHash presented a score below 10, the Mozilla Firefox and Google Chrome presented scores above 10 compared with Internet Explorer, as expected by the scenarios. Analyzing the results obtained by the *Perceptual Diff* algorithm, it was found that is consistent with results obtained by pHash, being possible to observe the large number of distinct pixels when screenshots were compared with Internet Explorer. Case 2 showed a different result than expected. The pHash algorithm assigned a score above 10 for screenshots of Mozilla Firefox and Google Chrome browsers, but it was expected a score below 10, due to the similarity of the images. Perceptual diff worked as expected, showing the similarity of screenshots browsers Mozilla Firefox and

Table 1. ADDII test cases results

Scenario	BrowserA	BrowserB	Phash score	Pdiff (pixels)	Total (pixels)	Result	Expected results
Case 1	Mozilla Firefox	Google Chrome	5	3755	636582	Ok	Ok
	Internet Explorer	Google Chrome	20	147729	636582	Error	Error
Case 2	Mozilla Firefox	Google Chrome	14	6706	807143	Error	Ok
	Internet Explorer	Google Chrome	17	20607	807143	Error	Error
Case 3	Mozilla Firefox	Google Chrome	5	39022	636582	Ok	Ok
	Internet Explorer	Google Chrome	17	348794	636582	Error	Error
Site 1	Mozilla Firefox	Google Chrome	7	66773	1169566	Ok	Ok
	Internet Explorer	Google Chrome	15	218039	1169566	Error	Error
Site 2	Mozilla Firefox	Google Chrome	6	213276	1738232	Ok	Ok
	Internet Explorer	Google Chrome	7	291147	1738232	Ok	Error
Site 3	Mozilla Firefox	Google Chrome	6	322894	2819295	Ok	Ok
	Internet Explorer	Google Chrome	14	1235330	2819295	Error	Error
Site 4	Mozilla Firefox	Google Chrome	1	88033	908424	Ok	Error
	Internet Explorer	Google Chrome	6	192627	908424	Ok	Ok
Site 5	Mozilla Firefox	Google Chrome	4	417848	1966032	Ok	Ok
	Internet Explorer	Google Chrome	27	828384	1966032	Error	Error
Site 6	Mozilla Firefox	Google Chrome	6	285333	2014163	Ok	Error
	Internet Explorer	Google Chrome	2	337863	2014163	Ok	Ok
Site 7	Mozilla Firefox	Google Chrome	2	92767	979755	Ok	Error
	Internet Explorer	Google Chrome	2	119258	979755	Ok	Ok
Site 8	Mozilla Firefox	Google Chrome	4	282375	1803528	Ok	Ok
	Internet Explorer	Google Chrome	21	1173685	1803528	Error	Error
Site 9	Mozilla Firefox	Google Chrome	13	1037731	2903274	Error	Error
	Mozilla Firefox	Internet Explorer	14	522682	2903274	Error	Ok
Site 10	Mozilla Firefox	Google Chrome	3	325485	1163376	Ok	Error
	Mozilla Firefox	Internet Explorer	4	321559	1163376	Ok	Error

Google Chrome and the high number of distinct pixels in comparison with Internet Explorer.

On the websites checking, the scenarios of Sites 1, 3, 5 and 8 showed the expected results, illustrating the problem of rendering in Internet Explorer. The scenarios of Sites 2, 4, 6, 7 and 10 showed different results than expected. This fact shows that mainly pHash algorithm was not able to identify small differences in pages such as changes in small letters and pictures or missing areas. This is due to the reduction of the image and use only low frequency in the comparison process. The Perceptual Diff was more assertive in these cases, highlighting the difference in pixels pages that showed the highest difference, which in some cases showed more than 50 % of pixels different from one browser to another, such as sites 4, 6 and 9.

The project is available for download and complete data analysis report in http://www.fei.edu.br/~plinio.aquino/ADDII/.

6 Conclusion

Based on the scenarios evaluated during the work, it was possible to prove the viability of the ADDII as a tool for visual deformations caused by the mismatch in the interpretation and rendering of HTML content. We have concluded that the goals of generating screenshots and use of algorithms for visual perception established early in the project were achieved.

The examples with low/medium difficult (medium to high differences between the screenshots) were successfully detected and represented about 50 % of total amount of tests created to identify distortions caused by the browser. When selecting a standard web browser, the user ADDII defines a base where your code has been tested, allowing comparison with other browsers, allowing different combinations.

It was also possible to observe the greater efficiency of the Perceptual Diff algorithm, which presented efficiently the non-equal pixels during the comparison. Algorithm pHash had failed in tests where the visual differences were in small areas, which makes it a not good choice for images with large viewable area. For future work, it would be interesting to further explore the Perceptual Diff algorithm, delegating to the ADDII user the task of defining the acceptable percentage of distinct pixels for a given test, allowing the user to choose the acceptable level of difference to your pages. Even was not the main reason of the research, compare the two algorithms was inevitable, adding value to the research.

The result also demonstrates the correct decision to use both algorithms in a complementary manner, as it enables the analysis of two different views.

Allow authentication pages for testing in secured pages and set the resolution of the browsers performs a test are also desirable improvements in future versions.

Acknowledgment. To FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) for financial support.

References

1. Project Spartan announcement. http://blogs.msdn.com/b/ie/archive/2015/01/22/project-spartan-and-the-windows-10-january-preview-build.aspx
2. Grosskurth, A., Godfrey, M.W.: Architecture and evolution of the modern web browser, 1–24 (2006)
3. MSDN GetElementById reference. http://msdn.microsoft.com/enus/library/ie/ms536437(v=vs.85).aspx
4. Eaton, C., Memon, A.M.: An empirical approach to testing web applications across diverse client platform configurations. Int. J. Web Eng. Technol. IJWET Spec. Issue Empir. Stud. Web Eng. **3**(3), 227–253 (2007)
5. Choudhary, S., Versee, H., Orso, A.: WEB DIFF: automated identification of cross-browser issues in web applications. In: 2010 IEEE International Conference on Software Maintenance (ICSM) (2010)

6. Zhu, J., Liu, X., Urano, Y., Jin, Q.: A novel WYSIWYG approach for generating cross-browser web data. In: 2010 International Conference on Computational Science and Its Applications, pp. 155–164
7. IMAGEMAGICK. http://www.imagemagick.org/script/index.php
8. Wang, Z., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P.: Image quality assessment: from error visibility to structural similarity. IEEE Trans. Image Process. **13**(4), 600–612 (2004)
9. PYSSIM. https://github.com/jterrace/pyssim
10. Stevens, M.: On collisions for MD5, June 2007
11. Lin, C., Chang, S.: A robust image authentication method distinguishing JPEG compression from malicious manipulation. IEEE Trans. Circ. Syst. Video Technol. **11**(2), 153–168 (2001)
12. Fridrich, J., Goljan, M.: Robust Hash Functions for Digital Watermarking Department of Electrical Engineering, SUNY Binghamton, NY 13902-6000
13. Kozat, S.S., Mihcak, K., Venkatesan, R.: Robust perceptual image hashing via matrix invariances. In: Proceedings of the IEEE Conference on Image Processing, pp. 3443–3446, October 2004
14. Shepherd, E.: Perceptual Hash Algorithm. http://pastebin.com/Pj9d8jt5
15. Hamming, R.W.: Error detecting and error correcting codes. Bell Syst. Tech. J. **29**. 147–160 (1950). http://www.caip.rutgers.edu/~bushnell/dsdwebsite/hamming.pdf
16. Krawetz, N.: A Picture's Worth … Digital Image Analysis and Forensics Table of Contents, pp. 1–31 (2007)
17. Yee, H., Corley, S., Sauerwein, T., Breidenbach, J., Foster, C., Tilander, J.: Perceptual Image Diff. http://pdiff.sourceforge.net
18. Daly, S.: Visible differences predictor: an algorithm for the assessment of image fidelity. In: SPIE/IS&T 1992 Symposium, vol. 1666 (1992)
19. SELENIUM. http://seleniumhq.org/