國立臺灣大學電機資訊學院資訊工程學系
碩士論文
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

SeeSS：爲 CSS 開發者設計之即時變更衝擊視覺呈現

SeeSS: Instant Change Impact Visualization for CSS Developers

梁翔勝
Hsiang-Sheng Liang

指導教授：陳彥仰博士
Advisor: Mike Y. Chen, Ph.D.

中華民國 103 年 10 月
October, 2014

國立臺灣大學
資訊工程學系

碩士論文

SeeSS：爲CSS開發者設計之即時變更衝擊視覺呈現

梁翔勝 撰

103
10

誌謝

誠摯感謝我的指導教授陳彥仰老師，指出了這個尚未解決的需求、未被完整探索的研究方向，並且不辭辛勞地透過討論，讓這份研究得以從概念付諸實行。老師在投稿 UIST 前對論文的細膩翻修，有如全能住宅改造王一般令人拍案叫絕，讓整份論文煥然一新、擲地有聲。感謝我的口試委員陳偉凱老師在口試時，指出了整篇論文論述時的微妙偏差，並協助調整論文的方向，也謝謝口試委員余能豪老師、陳昇瑋老師的修改建議，讓這份研究能夠更發完整。

2013 年初的一對一的討論中，老師提出寫 CSS 時都不知道會改到哪些地方的困擾，開啓了一段奇幻的投稿之旅。這段短短三個月的旅途中，實驗室的學弟們各顯神通：郭冠宏與林裕欽完成了追蹤並比較變更，與建立快照及縮圖重要核心元件；李柏緯徹夜趕出使用者測試的流程、待測網站、給受測者的 task，以及記錄使用者動作時間的工具；詹雨謙負責前端的界面，在投稿前更徹夜拼出 demo video，完美地詮釋了 SeeSS。在研究的第二階段，感謝邱詩雅與陳俊瑋的加入，讓 SeeSS 的報告頁面得以有現在的外觀。我想，未來的我一定會想念這些在實驗室的日子；這段日子縈繞著學長姐、學弟妹們的認眞討論，佐以機智且非常爆笑的閒聊與吐槽。謝謝實驗室的夥伴們讓實驗室一直保持著這樣的氣氛與熱度，和大家一起做研究眞的非常開心。

謝謝第一階段的受試者，以及第二階段 Brainstorming 的參與者，對 SeeSS 的寶貴建議以及有趣的新方向。也謝謝那些在收集開發資料的時候，被我用 facebook 與 IRC 叨擾的朋友們的包容。

最後，我要感謝我的母親，日日叮嚀並且關心實驗室待到凌晨的我。也謝謝還有關心我的畢業進程與未來規劃的同學們。是大家的鼓勵，使得這份論文得以完成。眞的非常感謝大家!

摘要

層疊樣式表（CSS）是表述網頁外觀呈現的基礎網頁語言。開發者將一份 CSS 樣式規則套用在數個頁面，減少原始碼的重複，並且讓網頁有一致的外觀。然而，當 CSS 規則被修改時，開發者也得要手動地、事必躬親地去檢測每一個被變更到的網頁區塊。我們推出 SeeSS，一個可以在網站裡自動追蹤 CSS 變更、並替開發者將所有變更視覺化的系統。SeeSS 會以漸變動畫來強調網頁變更前後的不同之處。SeeSS 的開發歷經兩個迭代，初期成果也已經以開源的方式，釋出給網頁開發者們。

關鍵字

開發工具；工具包；網頁開發；層疊樣式表；變更視覺化

Abstract

Cascading Style Sheet (CSS) is a fundamental web language for describing the presentation of web pages. CSS rules are often reused across multiple parts of a page and across multiple pages throughout a site to reduce repetition and to provide a consistent look and feel. When a CSS rule is modified, developers currently have to manually track and visually inspect all possible parts of the site that may be impacted by that change. We present SeeSS, a system that automatically tracks CSS change impact across a site and enables developers to easily visualize all of them. The differences before and after the change are highlighted using animations. We refine its implementation during a 2-iteration design process, and release SeeSS as an open source tool for all web developers.

Keywords

Development Tools; Toolkits; Web Development; Cascading Style Sheet; Change Visualization

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cascading Style Sheet (CSS) is a fundamental web language for describing the presentation of web pages, such as colors, fonts, and layout. It is primarily designed to enable the separation of content from presentation, and has been a W3C standard since 1996 [25, 44].

One of its key benefits is reusing the same CSS files and rules on the same page and across multiple web pages, which reduces repetition, increases download speed, provides a consistent look and feel, and improves maintainability. CSS reuse is a common practice. For example, on Apple and McDonald's US websites, each CSS file is referenced by an average of 22.8 and 6.1 HTML files respectively.

When modifying a CSS rule, developers need to first visually check that the intended change has been correctly implemented. Then they need to verify that the change does not introduce unintended bugs on all parts of the site that reference the modified CSS files and reuse the modified CSS rules [22]. Figure 1.1 shows an example where a simple change to fix an alignment bug of the Login button on one page breaks the layout of other buttons elsewhere on the site.

CSS is supposed to be a simple, declarative collection of style rules. However, despite the fact that CSS is essentially not a programming language, CSS authoring is often regarded as "programming" [22, 36] and it is necessary to debug CSS code [35]. Web developers have to deal with sophisticated cascading rules, take care of selector specificity and watch out for selectors matching unwanted HTML elements. Even the CSS 2.0 specification itself is too complex for people to understand how CSS features interact [2].

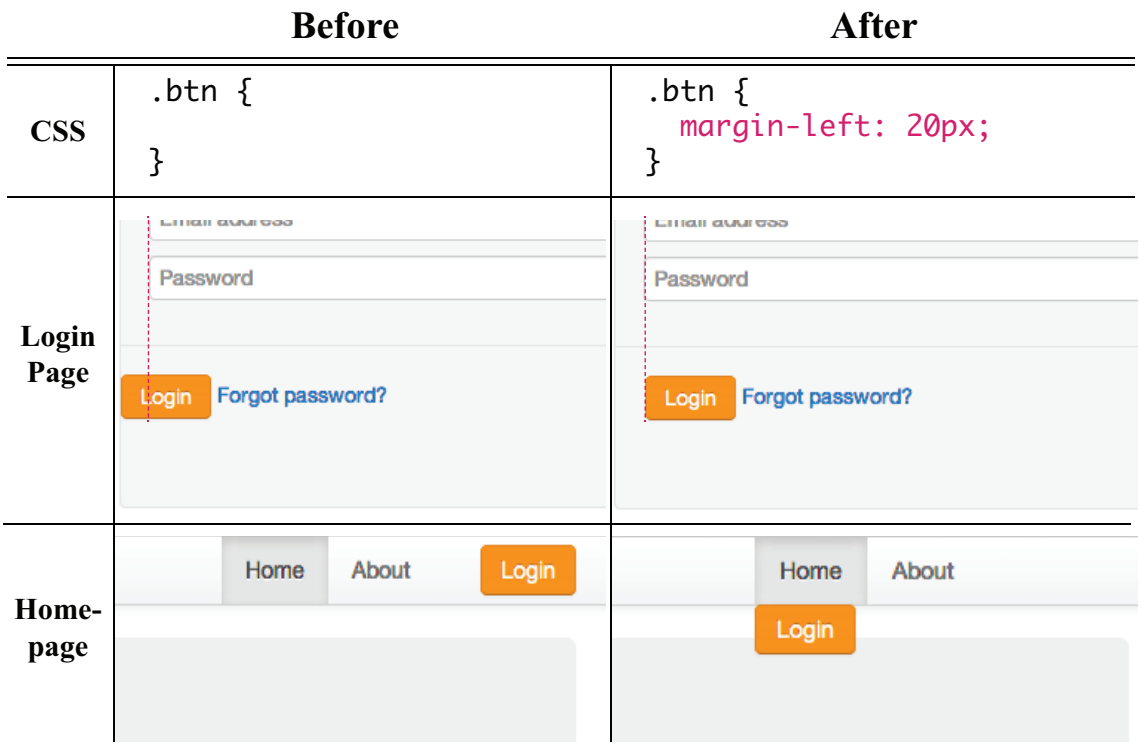

Figure 1.1: Modifying a CSS rule may fix a bug on the current page, but introduce bugs elsewhere: a developer fixes the "Login" button alignment by adding a `margin-left` property to the login page; however, such change breaks the "Login" button layout in the navigation bar on the home page. This bug is not immediately visible to developers using currently available tools.

Last but not least, visual properties of an element can come from rules scattered across multiple style sheets. It is often not clear how a modification of CSS code will impact the representation of a document [35].

Many tools have been developed for web programming: inspector tools such as Firebug [18] and Chrome and Safari's built-in tools [12] help developers inspect rendered web pages individually, but do not help track changes across a site. Web UI testing tools such as Selenium [41], Sahi [40], and Sikuli [45] support UI automation when performing functional testing of web pages, but may not detect non-functional problems introduced through CSS changes. Mogotest [30] is a web-based regression testing tool that visualizes changes on pages individually, but do not support interactive development nor multi-page visualization.

We present SeeSS, a system that tracks CSS change impact across a website and visually presents parts of the site that have changed due to the CSS code modification. As

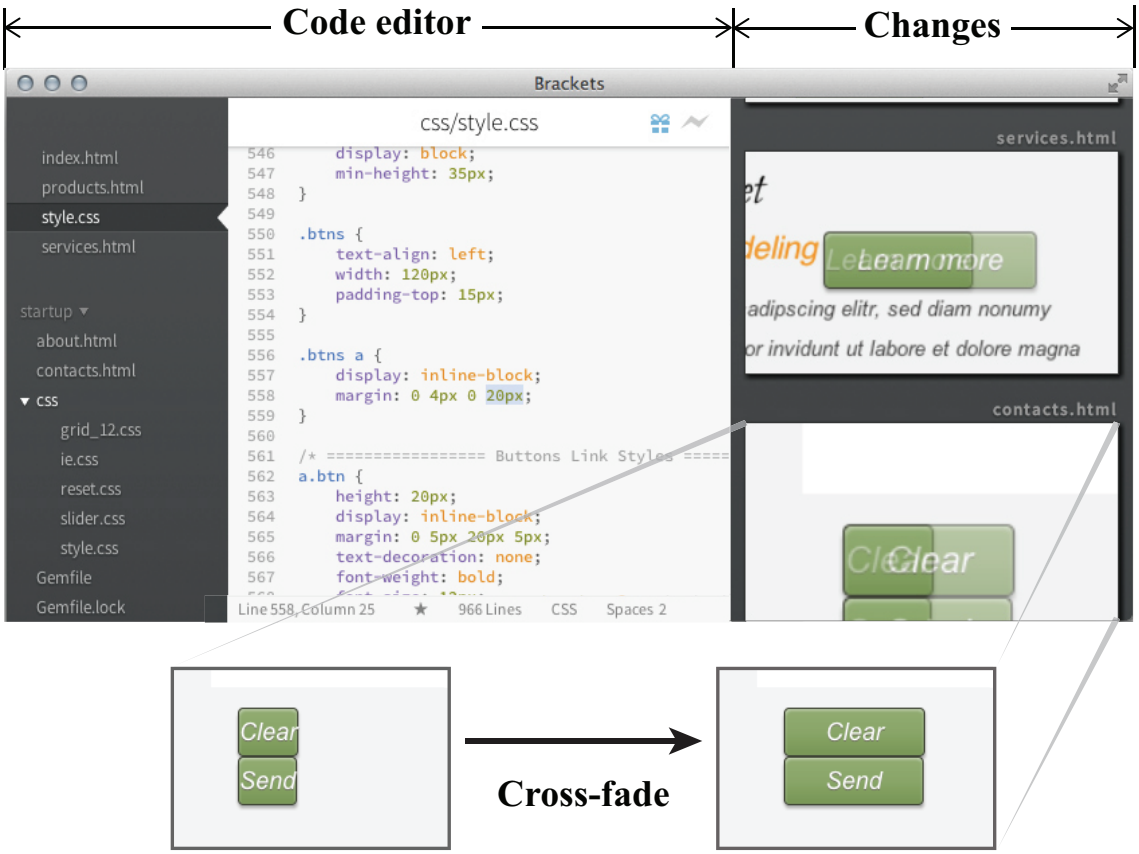

Figure 1.2: SeeSS graphical user interface: the left panel corresponds to the code editor, and the right panel shows a list of visual changes associated with the most recent CSS modifications.

shown in Figure 1.2, SeeSS computes and shows all visible changes before and after the modification. It also renders and stores snapshots of all pages across the website, and compares them across revisions.

We developed 2 SeeSS prototypes, one after another, under an iterative design [31] process. In the 1st iteration, we develope a SeeSS prototype that tracks HTML and CSS references of a static site when documents are opened and saved. A small lab-based user study shows promising initial results. All of the 4 participants reported that they were faster at fixing CSS problems with SeeSS and all would like to continue using SeeSS as their regular web development tool. The 2nd iteration starts with a brainstorming session with 8 web developers. The 2nd prototype improves the performance and applicability across different types of websites, aiming for public release. The evaluation of this iteration is currently ongoing.

We reported our initial results of the 1st iteration in a research paper, 'SeeSS: Seeing

What I Broke – Visualizing Change Impact of Cascading Style Sheets (CSS)', which was accepted for presentation at and publication in the proceedings of the 26th annual ACM symposium on User interface software and technology [24].

This thesis reports the design process of SeeSS so far. We first compare related web developer tools and previous researches in web testing in Chapter 2. We then describe the design and implementation of SeeSS in Chapter 3. Afterwards, we present our preliminary user study process in Chapter 4. The first iteration ends as we discuss the user study results in Chapter 5.

In the second iteration we first explore posibilities of presentations of change impacts via a brain-storming session with 8 web developers in Chapter 6. Based on that, we then describe the design choices in Chapter 7 and what we choose to leave out in Chapter 8. Lastly, we conclude this thesis with a summary of the 2 iterations in Chapter 9.

Chapter 2

Related Work

2.1 Web Browser Developer Tools

Modern desktop browsers like Google Chrome, Safari, Firefox, and Internet Explorer have built-in inspectors [12, 18] that help developers view, debug, inspect, and profile individual web pages [19]. The Document Object Model (DOM) inspector enables developers to select specific elements on the current page, view all the CSS attributes applied to these elements and locate the CSS files containing each associated CSS rules. While the DOM inspector can help developers locate and modify a particular CSS rule, developers still need to manually track all HTML files that reference the corresponding CSS file, and check all parts of the site which are impacted by a modification of this rule.

2.2 Web UI Testing Automation Tools

Selenium [41] and Sahi [40] are designed for automating functional tests of web pages. Pre-recorded input sequences, such as mouse and keyboard events, are fed to the website, then assertions are used to check if responses matches expected patterns (e.g. if certain text exists). Sikuli [7] takes a visual approach: screenshots are used to depict input sequences and help visually check the resulting patterns.

While these tools help test the functionality of websites, they are not designed to support CSS programming. First, in the case of dynamic web applications, some of the visual

programming may happen independent of the functional programming. For example, the HTML/CSS/Javascript may be developed by front end developers, while the functional programming is done by the back end developers. Second, the desired states for CSS are often described by wireframes before the final visual mockups are ready. SeeSS facilitates iterative CSS development without a functional website and before the final mockup of the site is ready.

2.3 CSS Unit Testing Tools

The developer community has rolled out several tools regarding CSS unit testing, hoping to make CSS as testable as Javascript or server-side applications. Such tools like CSSunit [11], CSSert [10], Cactus [6] and Hardy [20] take a test script as input. Developers may specify in the test scripts the expected value of a certain CSS property of an HTML element. Figure 2.1 shows a specification that asserts <p> elements to have gray color. However, in order to clearly describe visual appearance of an element, developers would have to extend the number of assertions to a certain extent. It requires much more work than functional unit tests, whose code under test usually contains just a few functionalities to test upon. The rapid changing nature of user interfaces also increases the maintenance cost of CSS unit test cases.

```
Feature: Website layout test
As a user I want visual consistency on the http://csste.st/ website

Scenario: Content layout
Given I visit ”http://csste.st/”
Then ”section␣>␣p” should have ”color” of ”rgb(68,␣68,␣68)”
```

Figure 2.1: An example of Hardy [20] test script[1].

[1] The example was directly excerpted from Hardy “Getting Started” documentation `http://hardy.io/getting-started.html`.

2.4 Regression Testing for Web

Regression testing shows change impact and ensures that the intended behavior is preserved after modification [17]. While regression testing has been successfully applied in many domains, regression testing web applications is notoriously daunting due to the dynamism in web interfaces. Experiments shows that 82% of test case output differences are false alams [38, 42]. Previous work on web regression testing for web mainly focus on testing functionalities, in which the test cases are comprised of sequences of events (e.g., "click on button", "enter text") on GUI widgets (e.g. "button", "text-field") [3]. A test passes if and only if the event sequence of a test case can be replayed on the modified web application. In order to reduce false alarms of functional regression tests, previous researchers [37, 38] ignore discrepancies that are visible to human but do not affect functionality. In contrast, SeeSS targets the visual differences and is not designed to detect the functional changes.

Huxley is a test-like system for catching visual regressions in Web applications [21]. It runs in 2 modes: in *record mode*, developers can take screenshots of the website under test as they go through the functionality they want to test. Before submitting code for review or in continuous integration, developers should run Huxley again in *playback mode*, which re-runs the user actions, takes new screenshots and compares them against the old ones. The detected screenshot differences will show up in a commit in the source code repository for the UI designers to review the differences afterwards.

Mogotest [30] is a web-based consistency testing service that provides regression testing across multiple browsers. It compares the current DOM snapshots of the site with previous versions. For each page, it shows a side-by-side view of two versions if it detects a change. Instead of visualizing individual and entire pages, SeeSS is designed to identify the exact set of regions that changed and to enable users to visualize all of them at once. This enables developers to verify the intended changes and to spot any unintended changes at a glance.

Lastly, Huxley, Mongotest and other similar tools [9, 33] collects the visual discrepancies aggregated in a period of time in a tedious report. Developers need some effort to

recall and track down the suspicious edits when they receive the error report long after the actual modification. In contrast, SeeSS analyzes and visualizes changes whenever a file is saved in order to provide immediate feedback, revealing errors before they accumulate.

Chapter 3

First Iteration Design and Implementation

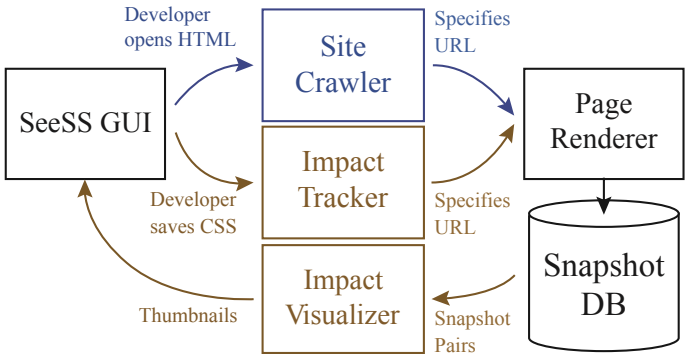

Figure 3.1: The 1st iteration SeeSS system architecture and data flow.

In order to support the tracking of CSS change impact, SeeSS needs to: (a) track pages that are part of the current site, (b) detect all visual changes across the site due to a CSS modification, and (c) visualize the differences. Figure 3.1 shows the system architecture and data flow.

3.1 SeeSS GUI

The SeeSS GUI supports editing of HTML and CSS files, listens to HTML and CSS file open and save events, and displays the rendered visual changes as a series of *thumbnails*. A thumbnail represents a rendered page region that has visible differences between the

current and the last snapshots. It consists of two cropped snapshots representing how the region looks *before* and *after* the CSS change.

The thumbnails support the following two user interactions: (a) when the mouse hovers over the left half of a thumbnail, the animation stops and the previous version of page fragment is shown. When the mouse hovers over the other half, the current version is shown; (b) clicking on the thumbnail opens the corresponding HTML document in the editor window.

3.2 Site Crawler

When an HTML page is opened in the GUI, the file is passed to the Site Crawler, which performs a recursive traversal of the hyper-links in the file. In addition to calling the Page Renderer to take initial snapshots for each crawled page, the crawler also records all CSS files that are referenced by each page. The dependency graph between the HTML files and the CSS files is used to track change impact.

3.3 Impact Tracker

When a CSS modification has been saved, the Impact Tracker looks up the set of HTML files referencing the changed CSS file using the dependency graph. It calls the Page Renderer to render the HTML files and save the snapshots to the database, and then calls the Impact Visualizer to detect differences.

3.4 Impact Visualizer

For each HTML file that references the changed CSS, the Impact Visualizer fetches the current and the last snapshots and calculates their visual differences. Because a CSS modification may impact multiple regions of a page, bounding boxes are calculated to capture the areas where the two versions differ. The bounding boxes are used to crop the snapshots and the results are displayed in the GUI. An animation is used to help highlight the visual

differences to developers. Details about the animation are given in the implementation section.

3.5 Implementation

We implemented the SeeSS GUI using the Brackets open-source text editor [5] with a custom SeeSS extension. The extension is implemented using HTML, CSS, and Javascript. It is a resizable sidebar on the right side of the editor window. It displays the visual differences using CSS cross-fade animation, a common approch used to smoothly transit between two states [4, 15].

The other components are implemented using Node.js v0.8.8 [32], a Javascript-based server framework, running on Mac OS X 10.7. They communicate with the SeeSS Brackets extension via socket.io, a Javascript library for real-time communication. PhantomJS v1.9 [34], a headless WebKit rendering engine, is used to render the HTML pages. A pool of PhantomJS processes is used to crawl the site and to render multiple pages in parallel. The snapshots are full-page screenshots captured using PhantomJS.

The current implementation of the impact ranking algorithm is the mean square error of the RGB values of the pixels between the two versions, which is the sum of squared difference of each pixel's red, green, and blue values, divided by the area of each bounding box.

Chapter 4

First Iteration Developer Feedback

We conducted a small lab study to understand how SeeSS supports web developers' workflow [23]. Given the small number of participants and the large number of potential factors that could affect task completion time and error, our goal was to collect feedback rather than quantitative comparisons.

4.1 Study Design

We created two websites with CSS problems, and asked each participant to debug one site then the other. Figure 4.1(b) shows one of a CSS problem statement we showed to the participants. The study used a within-subjects design; the order of the websites and the order of whether SeeSS was used were counter-balanced. We asked the participants to think-aloud while fixing the CSS problems. Afterwards, participants filled out questionnaires, and discussed their experience in semi-structured interviews.

Each site had 5 CSS tasks that the participants needed to complete. The tasks were designed to cover the following CSS programming challenges: (T1) modifying rules such as font-family and layout widths that are typically used throughout the site; (T2) modifying rules that apply to elements with multiple occurances on the same page, such as the appearance of buttons and widgets; (T3) tinkering with `position` and `z-index`, which are affected by attribute settings of ancestor elements; (T4) overriding existing rules, which requires the knowledge of selector specificity, style cascading, and style inheritance; (T5)

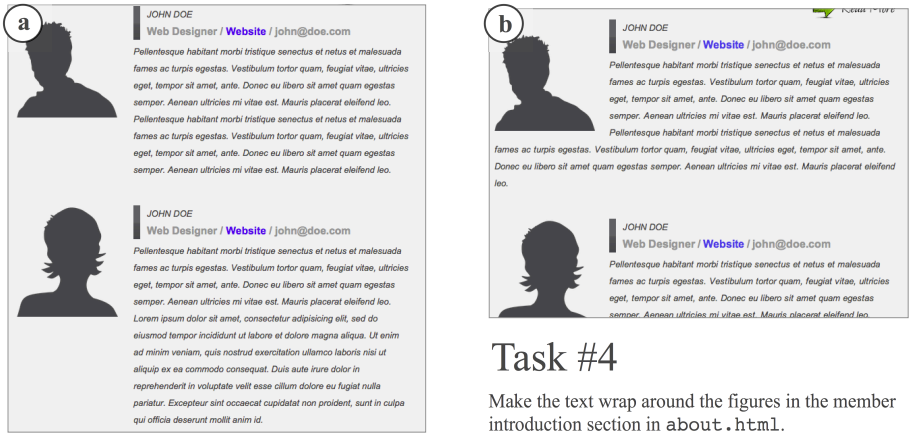

Figure 4.1: (a) The original look of one of the websites presented to the participants. (b) The corresponding CSS problem statement shown to the participants during the user study. The problem statement includes a screenshot of what the participants should achieve, followed by the text decription of the task.

avoiding ineffective modifications, such as applying `height` attribute to an inline element.

The experiment setup mimicked typical web development environment, with a code editor to make edits and a browser window to visually check and inspect the results. The experiment was carried out on a Mac Mini equipped with Intel CoreDuo 2.53GHz and 8GB DDR3 RAM.

4.2 Participants

After a pilot study with 2 users, we recruited 4 participants (1 male) on average of age 22 who had CSS programming experience from our university. The participants self-rated their HTML and CSS proficiency between 3 and 4 on a 5-point Likert scale (average 3.3) with 5 being an 'expert'. The participants used Sublime Text and Vim as their primary code editor, and none had experience with the Brackets editor.

4.3 Results

The post-experiment questionnaire and interviews showed positive perception and promising results. On a 5-point Likert scale indicating levels of agreement, all four participants

rated SeeSS a 4 on helping them fix CSS problems more quickly. They also all rated a 4 when asked if they would use SeeSS as their regular web development tool.

However, when asked if SeeSS' speed was acceptable, three participants rated it 3 and one rated it 2. We analyzed the system logs and found that SeeSS took an average of 5.7 seconds to display the thumbnails (standard deviation was 4 seconds), which was significantly slower than the sub-second rendering time of a browser page reload.

Chapter 5

First Iteration Discussion

5.1 Waiting for Thumbnails

We observed that participants sometimes came to a short halt after saving a file. When asked about what they were waiting for, participants reported that they were waiting for the SeeSS thumbnails to display. This was a behavioral change owing to the latency of SeeSS, but also suggested that SeeSS provided sufficient utility for participants to be willing to wait. In any case, SeeSS' speed should be significantly improved and the user interface should provide a progress indicator.

5.2 Using File Search

P2 and P3 used the "Find in Project" command to look at all the HTML files that referenced the modified CSS file. Interestingly, they only used it when SeeSS was not available, and never used it when SeeSS was available. They reported being cautious about making edits because the code was written by someone else and the impact of each CSS rule was unknown. The situation is similar for collaborative projects, especially for new team members who are not yet familiar with the dependencies among various HTML and CSS files. SeeSS can play an important role in helping developers become familiar with new projects and avoid breaking other developers' code.

5.3 Designing Thumbnails

Pilot 1, P4 and P2 reported that the thumbnails help by providing a holistic view of all changes across the website. Pilot 1 specifically mentioned that the holistic view provides essential information to determine whether he should modify a CSS definition or create a new CSS class.

Two of the other participants reported difficulty in matching elements in the thumbnails to pages they are viewing in the browser. They suggested limiting the thumbnail scaling to a maximum of 1:1, and show more surrounding elements to provide more context when possible.

P4 reported that the cross-fade animation conveyed the concept of "change", while P1 and P2 would like the thumbnail to be static, showing the "after-change" state of the website. P3 and Pilot 2 could not tell "before-change" and "after-change" from the cross-fade animations. Scaling and translation animations may be a better alternative to cross-fade when visualizing dimension or position change.

When an element in the upper part of the page increases its height, it shifts downwards all the elements that comes after. SeeSS generates one thumbnail for each of those affected elements, where some participants found it too verbose. To make things worse, when a tall element shifts vertically, SeeSS will detect two separate changes: The movement of top edge results in one thumbnail, and the movement of bottom edge results in the other. As a result, thumbnails often come in a large number, and look duplicated. The participants needed to scroll through the list to look for interesting changes.

All the participants commented that the most important thumbnail was at times not the first on the list. P2 and P3 suggested grouping thumbnails from the same page together and sort them by their y-position on the page. This may also make it easier to locate elements in the document. Clustering similar changes may also help summarize the thumbnails.

5.4 Summary of First Iteration

In the first iteration we implemented SeeSS GUI that integrates a code editor with thumbnails visualizing code changes. It provides developers a holistic view of changes across all pages of a static website.

We conducted a 4-person preliminary study to evaluate SeeSS. All the participants agreed that the visual feedback SeeSS provides right after file saves helps them fix CSS problems more quickly. We also observed that SeeSS introduced interesting behavior change to the participants. It provides useful visual feedback so that the participants were willing wait for the thumbnails to display. Two participants stopped using "Find in project" function in the code editor, preferring SeeSS notifying them the impact of the modified CSS rules. To sum up, a change impact visualization tool such as SeeSS can be helpful in CSS development. It deserves further polishments; in particular, it is the processing speed that affects the experience the most.

Chapter 6

Second Iteration Brainstorming

6.1 Empirical Design Choices in Previous Iteration

The preliminary user study in the 1st iteration shows that it can be helpful to CSS developers when we provide change impact as a feedback during development. Before the 2nd iteration, we step back and review the empirical design choices we have made previously, seeking for alternative solutions and room for improvement. The design choices include:

- We use thumbnails to visually represent each individual change impact.
- SeeSS GUI combines the change impact feedback and a code editor.
- We use animation to highlight visual differences.

In order to find out if the empirical choices fit web developers' usage pattern, we held a brainstorming session with a small group of web developers. The developers shared with us their developing experiences, and showed us what a tool like SeeSS should be from their perspectives.

6.2 Brainstorming Procedure

Our brainstorming session consisted of 3 consecutive phases. In phase 1, we asked the participants to describe the tools and window arrangements they use when developing

Figure 6.1: During the brainstorming session, a participant is sharing an idea and describing how his idea can be applied to a code editor window on a screenshot image sticked to the whiteboard. This footage is taken from the actual recording of the session.

websites. The participants then shared with each other some visual errors they experienced when authoring CSS. We then introduced the brainstorming principles to the participants.

In phase 2, we imagined a system that could precisely find out errors and mistakes as soon as a modification is made to CSS code. We asked the participants how such a system should present the errors so that developers can quickly identify and fix the relevant part of code. The participants could write down their ideas on sticky nodes and post them on a whiteboard. To help solidifying the ideas, the participants were encouraged to come up with ways to describe or represent the CSS errors previously shared in phase 1. Also, we displayed a screenshot of a code editor and a browser window (as in Figure 6.1) during the session, reminding the participants to ideate for a developing scenario.

Phase 2 was a warm-up for the next, real problem-solving phase. In phase 3, we degenerated the imaginary system in phase 2. Instead of an omnipotent system that differentiates errors from correct changes, we now have a system, which is more like SeeSS, that just reports all the visual changes and cannot point out the errors. We asked the participants how such system should present the changes so that developers can quickly tell if the modification is correct.

Lastly, we asked the participants to vote for up to 5 favorite ideas.

6.3 Resulted Ideas and Concepts

We invited 8 CSS developers (4 females) to participate the 2-hour brainstorming session. In phase 1, one of the participants reported working in a 3-screen setup, with source code and web browser arranged in separate screens. All the others said they constantly switching between the code editor and the web browser after every CSS file save. When we went through all the 3 phases, The participants placed 29 votes on 48 ideas. After the brainstorming session, we categorized the 48 ideas into 11 concepts. An idea can have one or more concepts embedded. Some concepts conflict with the other concepts. For example, a concept that suggests collecting all changes in a central list would conflict with another concept that suggests showing the changes in where the changes are. We never categorize an idea into two conflicting concepts. The 11 concepts are described below.

Central Change List and *Changes in Context*

Central Change List and *Changes in Context* are two conflicting concepts. The former suggests putting all changes in a central list, while the latter shows the changes on where the changes actually happen. Figure 6.2 (a) and (b) are examples of *Central Change List*. (c) and (d) are examples of *Changes in Context*.

In-Browser and *In-Editor*

Some of the ideas specified where an individual change or a list of changes to be displayed. While most of the participants usually switch back and forth between browser windows and code editor windows, it was intuitive for them to come up with ideas that associates with these two applications. In Figure 6.2, (b) and (d) show the changes in code editors, (c) in browser windows, and (a) is an idea that applies to both the browsers and the code editors.

It is worth noting that the participants regarded code editors as “input” panels, and

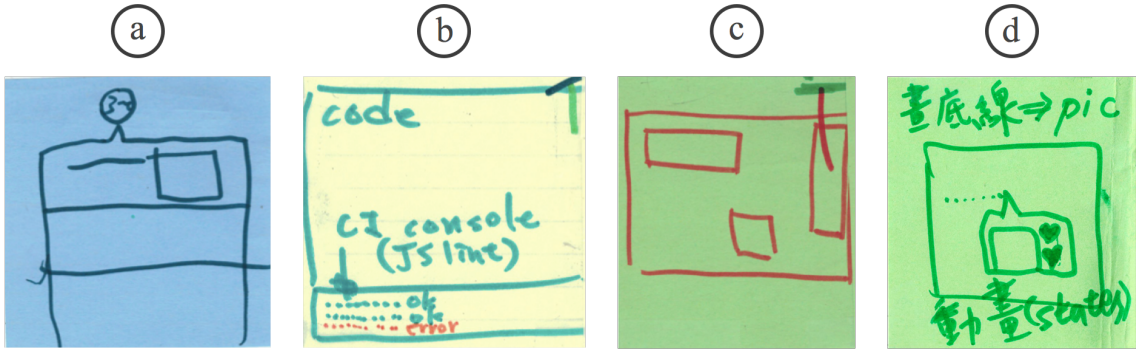

Figure 6.2: Four idea sticky notes from the brainstorming session. (a) Facebook-like list of changes. Embeds the concept *Central Change List*. (b) Changes listed in text displayed at the bottom of a code editor. Embeds the concept *Central Change List*, *Narrative Changes* and *In-Editor*. (c) Visual changes emphasized in a browser window. Embeds the concept *Changes in Context* and *In-Browser*. (d) Underlined code modifications and showing thumbnails on mouse hover. Embeds the concept *Changes in Context* and *In-Editor*.

browsers as where the "output" comes out. It is intuitive for the developers to see the outcome of a code change coming through browser windows.

Code Property Ranking, *Rendered Property Ranking*, *Code Property Clustering* and *Rendered Property Clustering*

In the brainstorming session, participants soon realized that there mights be lots of changes to display, and it may be necessary to rank or cluster the changes. *Ranking* concepts contains ideas that proposed a sortable metric. *Clustering* concepts are those suggested grouping changes together using a specific method. Ranking and clustering can be done by methods involving either source code properties, such as source code file name or commit histories, or rendered element properties, like if a CSS rule is overridden or if the elements are aligned with other elements (e.g. aethetic metrics.)

Animated Visualization

Ideas suggesting the system should emphasize the change using transition animations. This concept matches one of the empirical design choices we have made in the 1st design iteration.

Boolean Indicator

Ideas that notifies the developer of changes using indicators. For instance, an idea suggesting that the system should show a green dot after the source code file name in the code editor sidebar menu, if a style definition written in that file is changed or erred.

Narrative Changes

One of the participants came up with a series of ideas that require visual changes to be represented in text rather than being visualized. Those ideas are all categorized under this concept.

The proposer of those ideas believed that visual changes could be described in text in manner of the size, color and position of the visually changed elements. Afterwards, the changes can be shown in console window in text like in Figure 6.2(b), or even be heard by developers using text-to-speech programs.

6.4 Voting Results

The idea receiving the most votes (6 votes) suggested that all visual changes should be displayed in a grid-style fashion inside a browser window, like the "Top Sites" page in Safari browser (Figure 6.3.) The visual changes from different parts of the website are to be collected and displayed at once inside a grid layout.

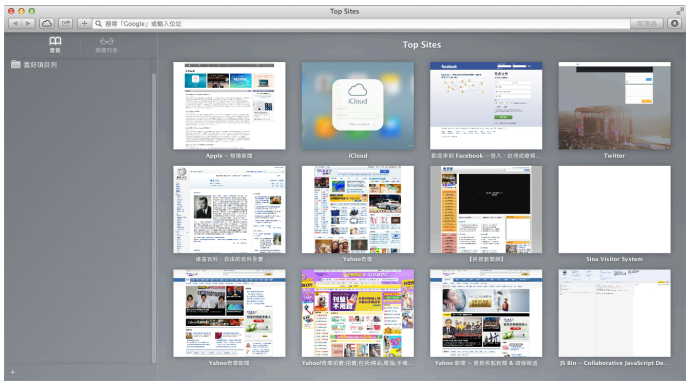

Figure 6.3: "Top Site" page in Safari browser. Using such layout to display visual changes is the most supported among all the other ideas.

Central Change List	9 votes	3 ideas
In-Browser	8 votes	7 ideas
In-Editor	4 votes	9 ideas
Changes in Context	4 votes	7 ideas
Animated Visualization	4 votes	2 ideas
Rendered Property Ranking	3 votes	2 ideas
Code Property Ranking	2 votes	4 ideas

Table 6.1: Popular idea concepts ranked using vote counts.

The second place goes to two ideas that has 3 votes each. One suggested that the system should highlight the visual differences using transition animations, whose concept was already adopted by SeeSS 1st iteration prototype.

The other idea proposed a method that ranks visual changes using the number of of the changed elements showing up inside the website. Changed elements with more occurrences should appear in the front of the change list. If an element is placed in a website for many times, this idea assumes any visual change to that element can probably introduce a large impact to the website and thus deserves human inspection.

We omit the rest because there are too many ideas with 2 votes or 1 votes.

Beside sorting individual ideas using vote count, we also calculates the sum of the vote count inside each concepts. The concepts with most votes inside are shown in Table 6.1. We also include the idea counts of these concepts to show how often each concept was brought up during the entire brainstorming session.

Central Change List is the concept that received the most votes, despite its conflicting concept, *Changes in Context*, was mentioned in more ideas. More participants preferred the concept *In-Browser* than *In-Editor*. A few votes goes to *Animated Visualization*, but few participants came up with such idea. Instead, most ideas were about displaying only the "after" state, e.g. the appearance of changed elements *after* code modification. Lastly, the participants had more thoughts on *ranking* than on *clustering*, which was not displayed in the table.

	Animated Visualization	Central Change List	Changes in Context	In-Browser	In-Editor	Boolean Indicator	Rendered Property Ranking	Narrative Changes	Code Property Ranking	Code Property Clustering	Rendered Property Clustering
Animated Visualization	1	1	1	1							
Central Change List		1		1		1		1			
Changes in Context			5	2	1					1	
In-Browser					1				1		
In-Editor					1	2	2			3	
Boolean Indicator										1	
Rendered Property Ranking											
Narrative Changes											
Code Property Ranking											
Code Property Clustering											
Rendered Property Clustering											

Concept A | # of ideas belongs to both A and B | Concept B

Table 6.2: Number of ideas in which a concept coexist with another concept. For instance, there are 3 ideas that belong to both the concept *Code Property Clustering* and the concept *In-Editor*.

6.5 Concurrency of Concepts

Some concepts tend to coexist with other concepts within an idea. Table 6.2 shows the concurrence of each two of the concepts. When a system presents the changes in browser windows (*In-Browser*), there are many ideas suggest showing the visual differences in there respective position (*Changes in Context*), despite the fact that the idea combining the concept *In-Browser* and *Central Change List* gains the most vote. When a system presents the changes in editor windows (*In-Editor*), most of the concurrent concepts (*Changes in Context*, *Boolean Indicator*, *Narrative Changes*, *Code Property Clustering*) were ideas that suggest further integration with the code editor, rather than just having a "sidebar" that presents a list of thumbnails. Such ideas include drawing an underline directly under the code that produces visual changes, showing a green dot after files that causes visual changes, describe visual changes using text [1] right after the line of code that causes the change, and so on.

[1] Short text messages like "element too wide" that describes visual changes.

6.6 Summary

From the voting result of the ideas, 6 out of the 8 participants preferred a centric list of thumbnails of changes put in a grid layout inside a browser window. Also, from Table 6.2 the participants did not expect a centric list of thumbnails put right inside an editor window. The brainstorming results suggest a redesign from the ground up, and point us a new direction with respect to change impact visualization presented inside a web browser window.

In regard to the empirical design choices we made in the 1st iteration, the brainstorming result supports that we use thumbnails to visually represent change impact. However, the result disagrees that SeeSS should display change impact feedback inside code editors. The participants did not care if the thumbnails are animated. In most of the idea they merely mentioned a static image of the elements after change.

Chapter 7

Second Iteration Design and Implementation

The review the developers' feedback in Chapter 5 indicates that the actual users of SeeSS would like it to be faster. Brainstorming session participants supported visualizing the changes in a grid layout in browser. Therefore, the main goal of the 2nd iteration is to create a tool that (a) provides "instant" feedback, and (b) shows the change impact visualization in a grid fashion inside browser.

Meanwhile, SeeSS in the 2nd iteration should still meet the 3 basic requirements from Chapter 3: (a) tracking pages in the current site, (b) detect visual changes, and (c) visualize the differences. Last but not least, we plan to release SeeSS to the community as an open source project at the end of 2nd iteration, benefiting as much web developers as we can. Therefore, we should consider the applicability of SeeSS to non-static websites such as single page applications. We should also reduce the effort for a developer to adopt such tool.

Bearing all that in mind, we drastically changed the design choice we have made in the 1st iteration, and also modified the overall structure of SeeSS. Figure 7.1 shows the new system architecture and data flow.

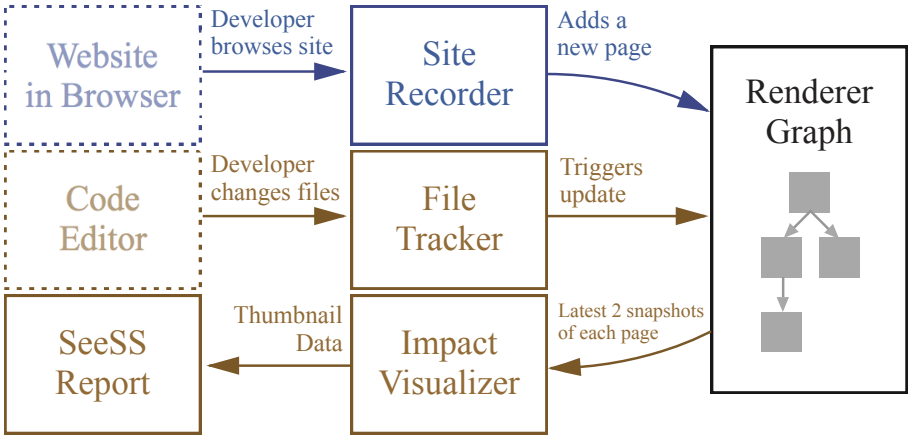

Figure 7.1: 2nd iteration SeeSS system architecture and data flow. Blocks with dashed borders participate in the data flow but are not parts of SeeSS.

7.1 Augment, Rather Than Replace Experiences

All participants in the brainstorming session reported that they would open up a code editor and a web browser at the same time when they write CSS. Making changes in editor and then switching to browsers for inspection has already become a routine in their CSS developing experiences. Interestingly, even though they would switch between different browsers to check cross-browser consistency, their code editor of choice does not get changed easily.

In the 2nd iteration, instead of pushing the users towards another IDE that supports SeeSS, we separate SeeSS from code editors so that adoption of SeeSS does not replace existing experiences. The File Tracker detects file changes directly from the file system. Therefore, developers can use any code editor to write CSS and HTML files. It looks sensible to give up on all editor visualizations and go for this choice, since participants in the brainstorming session like presenting visualizations in browser more than in editors.

7.2 Impact Visualizer

The Impact Visualizer is the only component left untouched from the 1st iteration. Still, it compares the latest 2 snapshots of each page and outputs the calculated thumbnail data as it did in the 1st iteration. It is the design of *snapshots* that we change the most.

In the 1st iteration, we used full-page screenshot image as snapshots of each page.

Comparing two screenshot images gives us the changes that are truly visible, hiding all the other changes that are irrelevant to SeeSS. Many existing CSS regression testing tools [9, 21, 33] also take the image-diff approach. However, after some benchmarking we found that the unsatisfactory delay of the 1st iteration prototype actually came from image processing. It took an average of 5.7 seconds to display the thumbnails. The PhantomJS headless browser took just 0.8 seconds to load the pages. But it took PhantomJS about 2.5 seconds to generate a full-page screenshot image whose height is a couple thousands of pixels, and the 1st iteration Impact Visualizer took another 2.5 seconds to calculate and generate the thumbnail images.

In contrast to the CSS regression testing tools mentioned above, previous work in web regression testing and cross-browser compability computes the difference using DOM tree properties after code modification [28, 30, 37, 38, 39, 42]. Similarly, the Impact Visualizer in 2nd iteration uses the DOM tree properties as snapshot of each page instead of screenshot images. Comparison of two snapshots involves traversing each node in the two DOM trees, determining DOM node mapping in between, and outputting the property values that differs. The Impact Visualizer also filters out the property changes that are not visible using a predefined blacklist of non-visual properties.

7.3 Site Recorder

In 2nd iteration, we extend SeeSS' applicability from static websites to dynamic websites powered by Javascript. Such dynamic websites can have many UI states not directly be accessible from the URLs.

The Site Recorder detects user interface changes, such as following links to another page, opening up a modal dialog or the application populating its content from external storage. A webpage after an such interface change is considered as a different "page" in SeeSS. The Site Recorder collects all the required information from a page so that SeeSS can reproduce such UI state afterwards. The collected page data, as well as the interaction needed to enter such UI state (for instance, mouse clicks on a certain element), is then passed to the Renderer Graph, where a page is taken snapshot and stored.

Enumerating UI states has been well researched in the field of automatic GUI test case generation [3, 8, 14]. Web UI automation techniques such as Crawljax [29] could be used to enumerate possible UI states and perform javascript-based navigation. Crawljax is also used in the regression testing of AJAX applications [37]. While automatic GUI test case generation and Crawljax excel at regression testing, large space of possible event sequences makes it difficult to be applied to an instant feedback tool like SeeSS. According to Crawljax's evaluation report [29], it would take Crawljax 14 seconds to enumerate all the 16 UI states on a small scale website.

On the other hand, capture/replay tools are the most popular tools for GUI testing [16, 27]. It operates in 2 modes: capture and replay. The former records the user action sequence, while the latter trys replaying the sequence. Instead of automatic tracking, we decided to take this approach so that SeeSS can provide instant change impact feedback on pages of developers' interest. The Site Recorder records each the user interface change as the developer clicks through the website under test after the developer activates SeeSS.

7.4 Renderer Graph

The Renderer Graph is the rendering and storage component of 2nd iteration SeeSS. It is composed of multiple renderers interconnected with user actions. Whenever a page is collected by the Site Recorder, its page data will result in a new renderer to be created inside the Renderer Graph. Each renderer is in charge of one specific UI state. The renderer takes and stores snapshots of that specific UI state, updates stored snapshots when the File Tracker detects CSS or HTML file modifications in the file system, then sends the latest two snapshots to Impact Visualizer after CSS or HTML file modifications. Finally, if an UI state can be reached from another UI state, there exists a directed *edge* from the later renderer to the former one. On the edge stores the details of the user action so that it can to be replayed later.

Figure 7.2 shows an example of how the renderer graph is constructed. As a developer activates SeeSS, SeeSS reloads the browser to make sure the initial UI state is accessible from URL. The reloaded page would result in a new renderer storing the snapshot of that

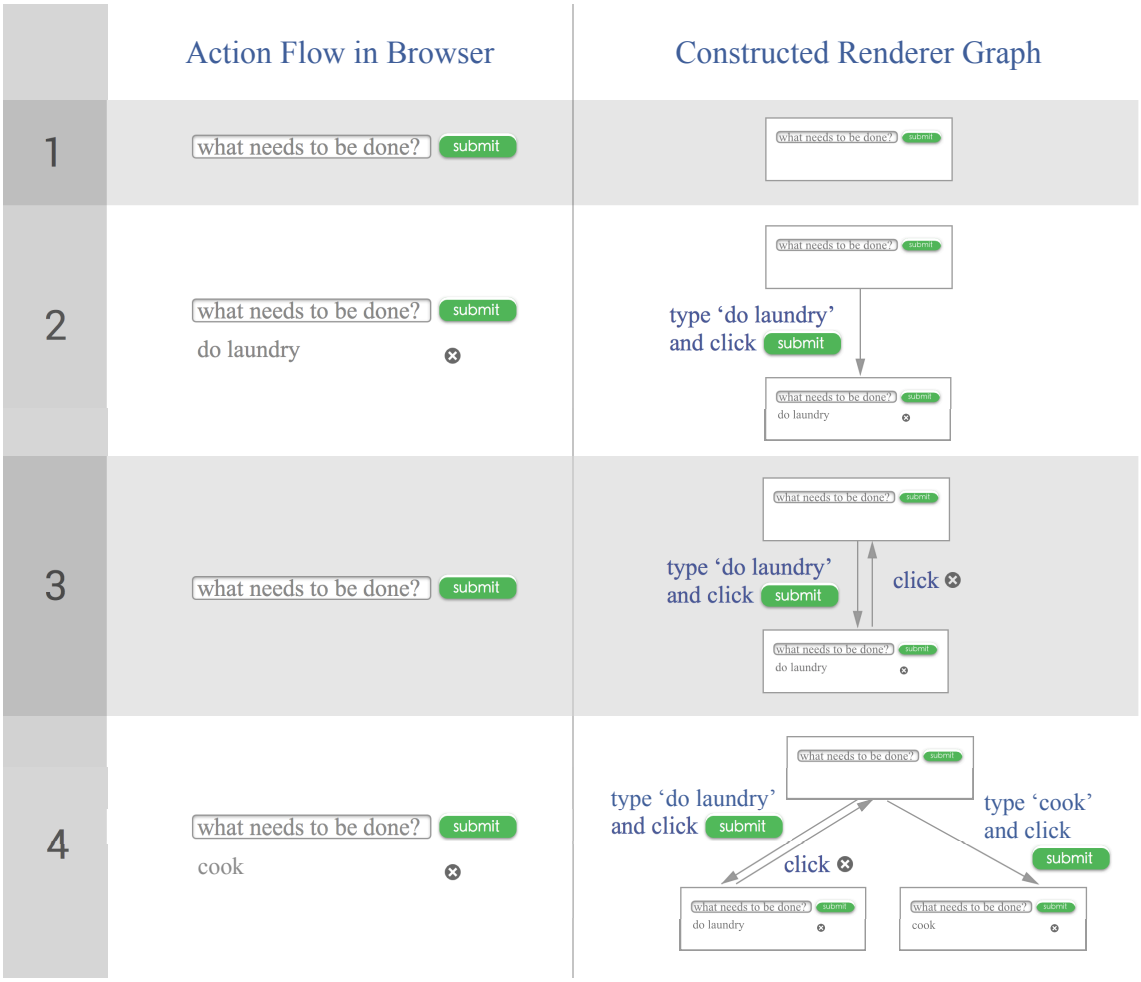

Figure 7.2: Constructing the Renderer Graph as the developer manipulates the To-Do list web application. Each step that updates the user interface would create a new renderer in the renderer graph. The developer performs the following steps, resulting in the renderer graph on the right: Step (1) the browser reloads and displays an empty to-do list. Step (2) developer submits a new item. Step (3) the new item is deleted by the developer. Step (4) developer submits another new item.

page. The developer then submits a to-do item in the list. This would cause an user interface change, enter a new UI state, and create a corresponding new renderer. There is also a directed edge from the first renderer to the second, remembering the click action and the clicked submit button. In addition, the Renderer Graph recognizes duplicated UI states (Figure 7.2 step 3), thus an UI state will not be assigned to two different renderers.

Figure 7.3 shows how to update the Renderer Graph after the developer makes an edit to HTML files. First, the Renderer Graph reloads all the renderers to fetch the new HTML document from the file system. Afterwards, starting from the root renderers of the renderer graph, the Renderer Graph updates the action target stored on the edges by

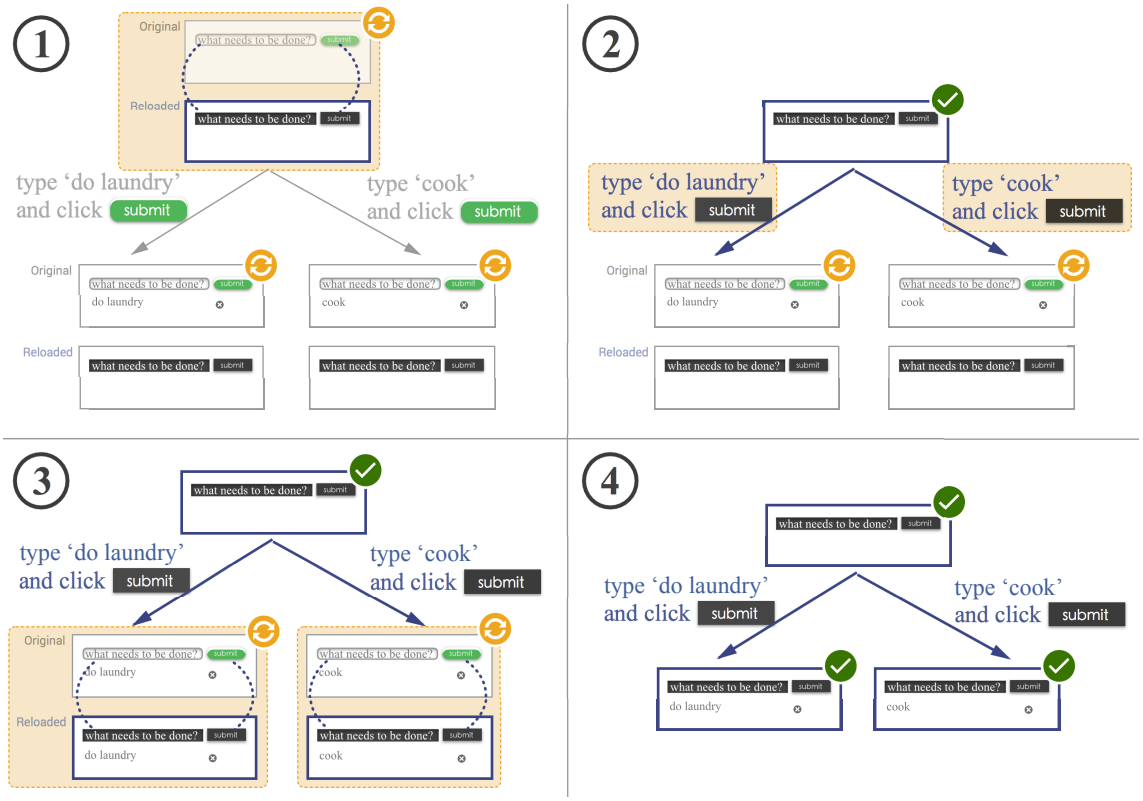

Figure 7.3: After developers adds an HTML class on the input and button element, the Renderer Graph updates all renderers in tree-level order. In the beginning, the Render Graph reloads all its renderers. Then (1) the root renderers calculates the mapping of elements between the two versions. The mapping is useful when comparing the snapshots and updating the outgoing edges (2). The Render Graph then replays the user action in the renderers pointed by the updated edges (3). As a result, all renderers in the Render Graph can be eventually updated (4).

mapping HTML elements before and after the refresh, replays the action in the non-root renderers, takes new snapshots, and then outputs the latest two version of snapshots, in a tree-level order.

When the File Tracker detects a CSS file change, the edges does not need to be updated so the situation is a lot simpler. The Renderer Graph only needs to update the CSS files linked in each renderers before taking new snapshots.

7.5 File Tracker

The File Tracker tracks the file system and inform the Renderer Graph of updating renderers upon HTML or CSS file modification.

7.6 SeeSS Report

The SeeSS Report is a tab in the browser where thumbnails visualizing change impacts are displayed in a grid layout.

7.7 Implementation

We implemented 2nd iteration SeeSS as a Google Chrome extension. The File Tracker acts as a LiveReload Client [26] connecting to the LiveReload servers pre-configured to watch the website file directories. Other components were all implemented inside the Google Chrome extension. The Site Recorder leverages DOM Mutation Observers to watch the user interface change. Renderers inside the Renderer Graph uses an HTML Iframe element to render and store page data. Lastly the SeeSS Report renders the animated thumbnails to a tab in Google Chrome (Figure 7.4).

We implemented Valiente's bottom-up algorithm [43] and the basic diffX algorithm [1]. SeeSS executes them one after another to map the HTML nodes between two snapshots inside the Impact Visualizer. It correctly maps the nodes even if there exists structural difference between the two DOM trees under a time complexity of $O(n_1 n_2)$, where n_i is the total number of HTML elements, text nodes and attributes of a DOM tree.

Lastly, 2nd iteration SeeSS is a open-source project initiated by us. The documentation and source code is available at `https://github.com/MrOrz/SeeSS`.

7.8 Early Results

We ran an informal benchmarking to measure the performance gain of the 2nd iteration SeeSS. The early results look promising; SeeSS in the 2nd iteration provides near-instant feedback, which is a great improvement compared to its predecessor.

SeeSS in the 1st and 2nd iteration were both provided with the same test website. The test website contains 6 static web pages. We made the identical CSS modification inside both systems. The CSS modification caused all buttons inside the website to become

Figure 7.4: SeeSS Report as a separate tab in Google Chrome. It highlights the changes to the dimension and position of an element using an transparent, animating red box. It also animates the properties like font and background color.

wider, and the both systems should generate same amount of feedback information. After the CSS modification is made, the 2nd iteration SeeSS took less than 1 second to visualize the change impact on SeeSS report, while the 1st iteration SeeSS took 5.3 seconds to generate the first thumbnail, and another 2.5 seconds to visualize all thumbnails. The processing log showed that it took the 1st iteration SeeSS 2.5 seconds to render and store full-page screenshot images of all 6 pages, and another 2 seconds to crop and generate the thumbnail images. It seems that image processing is the decisive performance factor between the two implementations.

Chapter 8

Limitation and Future Work

8.1 Changes not Recorded

The most obvious limitation of SeeSS in the 2nd iteration is that the change visualization is limited to the recorded pages only. Compared to the GUI automation approach such as Crawljax [29], the record/replay technique essentially fall short of a good coverage of the states, which is a trade-off for the processing speed. We plan to add a coverage report on how many possible actions are not included inside the Renderer Graph. For example, if a button exists in one of the recorded pages and is never pressed during the recording session, SeeSS should tell the developer about the button being omitted. With such report, the developers can have a better chance to discover omitted pages. Also, enumerating actions from the recorded states should more efficient than automatically and recursively traversing all the possible states.

8.2 Changes to the UI Flow

SeeSS cannot handle changes to user action's *behavior* after the developers modify Javascript. Such modification often introduces change to the UI flow, so that SeeSS cannot successfully replay the user action anymore. The developers would have to record the user action again after they make such modification.

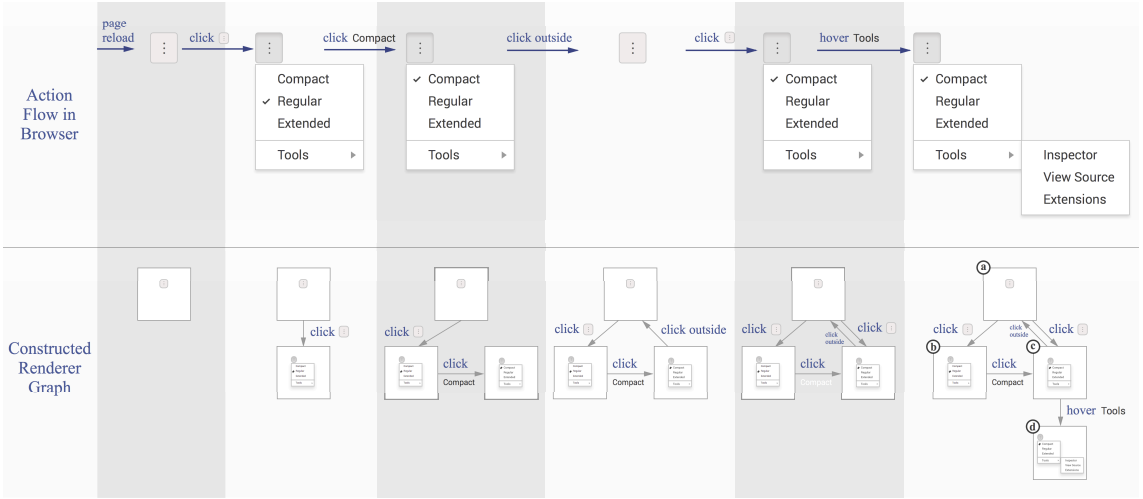

Figure 8.1: Constructing the Renderer Graph in a way that SeeSS cannot replay correctly. When SeeSS trys to replay the resulting Renderer Graph (on the bottom right corner of the figure) in tree-level order starting from renderer A, SeeSS would mistakenly regard renderer B as renderer C, thus report erroneous change impact. In the figure we assume that the state of the checked items (either 'Compact', 'Regular' or 'Extended') is stored in Javascript and does not affect DOM outside the menu, and that the DOM of menu is re-created from the stored state every time the '□' button is pressed.

8.3 Internally Maintained States

When updating the Renderer Graph, SeeSS traverses the renderers in tree-level order. If there is information that affects state transitions but does not introduce a new renderer in the Renderer Graph, it would confuse SeeSS when updating the Renderer Graph.

Figure 8.1 shows a scenario that confuses SeeSS. It consists of a menu that internally maintains the state of which menu item is checked. When the menu shows up every time, it relies on that information to put the check icon beside the correct menu item. Since SeeSS is not aware of the menu's the internally maintained state, when SeeSS tries to update the resulting Renderer Graph (right bottom corner of Figure 8.1) in tree-level order starting from renderer A, SeeSS would mistaken renderer B for renderer C, and incorrectly report that the position of the check has changed.

If the Renderer Graph were to traverse the renderers in the exact order of how the pages were captured, SeeSS would play well with the internally maintained states, at a cost of slower renderer traversal and hence the slower visual feedback. We plan to provide the developers with an option on how the renderers should be traversed in the preference

settings.

8.4 Thumbnail Animations

Our 1st iteration prototype supports cross-fade animation. DOM tree-based analysis, in addition to the original pixel-based approach, can be used to identify the change of dimensions, position and other properties of the individual DOM elements. In this way the accurate bounding box of each changed element can be determined by the element's DOM layout properties. Such an approach affords the support of a richer animation vocabulary that conveys complex changes such as text reflow, style change and glyph transforms in a similar fashion as so done in the markup code editor Gliimpse [15].

8.5 Impact Ranking and Clustering

Ranking change impact is still an open research problem that requires large-scale user studies and real cases of CSS bugs. Soechting et al model regression testing output that merit human inspection through a set of structural and syntactic features of two HTML documents [42]. Dobolyi et al displayed the faults of 800+ web apps to 386 people, asked them about the severity of each fault, then trained a model that determines the consumer-perceived fault severity [13]. It would be interesting if we build a similar human-perceived severity model by looking for bug fixes in open-sourced web applications, collecting code diffs from the commit log, then asking the users about severity, as well as analyzing the CSS properties relevant to those commits. Also, we plan to release SeeSS and collect some real statistics that can help provide insights to this problem.

Chapter 9

Conclusion

We have presented SeeSS, a system that automatically tracks CSS change impact across an entire website and helps developers visualize them. A preliminary, 4-person study showed promising results with participants all indicating fixing CSS problems more quickly with SeeSS, and all would like to use SeeSS as their regular web development tools. The development of SeeSS followed the iterative design flow, with significant difference in the design choice between the two iterations (Table reftab:conclusion). We are exploring alternative thumbnail visualizations, and human-perceived severity of change impact.

	1st Iteration	**2nd Iteration**
Design Goal	Working prototype	Performance & put visualization in browser
Where to report changes	In a sidebar of a code editor	In a tab of a browser
How differences are detected	Comparing the screenshot images	Comparing the DOM tree properties
Applicability	Static websites	Static and dynamic websites
Watched pages	All pages reachable using hyper-links	Recorded pages only

Table 9.1: Different design choices between the two iterations.

Bibliography

[1] R. Al-Ekram, A. Adma, and O. Baysal. diffx: An algorithm to detect changes in multi-version xml documents. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '05, pages 1–11. IBM Press, 2005.

[2] G. J. Badros, A. Borning, K. Marriott, and P. Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 12th annual ACM symposium on User interface software and technology*, UIST '99, pages 73–82, New York, NY, USA, 1999. ACM.

[3] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon. Graphical user interface (gui) testing: Systematic mapping and repository. *Information and Software Technology*, 55(10):1679 – 1694, 2013.

[4] A. Bezerianos, P. Dragicevic, and R. Balakrishnan. Mnemonic rendering: an image-based approach for exposing hidden changes in dynamic displays. UIST '06, pages 159–168, 2006.

[5] Brackets - open-source code editor. `http://brackets.io/`.

[6] Cactus. `https://github.com/winston/cactus`.

[7] T.-H. Chang, T. Yeh, and R. C. Miller. Gui testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1535–1544, 2010.

[8] W.-K. Chen, H.-T. Wu, J.-C. Wang, and Z.-W. Shen. An automatic gui testing approach using hierarchical finite state machine. In *The Fourth Taiwan Conference on Software Enfineering (TCSE08)*, 2008.

[9] CSS Critic: A lightweight framework for regression testing of CSS. http://cburgmer.github.io/csscritic/.

[10] cssert. http://thingsinjars.github.io/cssert/.

[11] CSSunit. https://github.com/gagarine/CSSunit.

[12] Chrome Developer Tools. https://developers.google.com/chrome-developer-tools/.

[13] K. Dobolyi and W. Weimer. Modeling consumer-perceived web application fault severities for testing. Technical report, 2010.

[14] S. Doğan, A. Betin-Can, and V. Garousi. Web application testing: A systematic literature review. *Journal of Systems and Software*, 91(0):174 – 201, 2014.

[15] P. Dragicevic, S. Huot, and F. Chevalier. Gliimpse: Animating from markup code to rendered documents and vice versa. UIST '11, pages 257–262, 2011.

[16] O. El Ariss, D. Xu, S. Dandey, B. Vender, P. McClean, and B. Slator. A systematic capture and replay strategy for testing complex gui based java applications. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 1038–1043, April 2010.

[17] R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. ESEC-FSE companion '07, pages 549–552, 2007.

[18] Firebug. http://getfirebug.com/.

[19] P. Geneves, N. Layaida, and V. Quint. On the analysis of cascading style sheets. WWW '12, pages 809–818, 2012.

[20] Hardy. http://hardy.io.

[21] Huxley. https://github.com/facebook/huxley.

[22] M. Keller and M. Nussbaumer. Cascading style sheets: a novel approach towards productive styling with today's standards. WWW '09, pages 1161–1162, 2009.

[23] T. D. LaToza and B. A. Myers. Designing useful tools for developers. PLATEAU '11, pages 45–50, 2011.

[24] H.-S. Liang, K.-H. Kuo, P.-W. Lee, Y.-C. Chan, Y.-C. Lin, and M. Y. Chen. Seess: Seeing what i broke – visualizing change impact of cascading style sheets (css). In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 353–356, New York, NY, USA, 2013. ACM.

[25] H. W. Lie. *Cascading style sheets*.

[26] LiveReload. http://livereload.com/.

[27] A. M. Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):4:1–4:36, Nov. 2008.

[28] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 561–570, New York, NY, USA, 2011. ACM.

[29] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. TWEB '12, pages 3:1–3:30, 2012.

[30] Mogotest | Web Consistency Testing Made Easier. http://mogotest.com/.

[31] J. Nielsen. Iterative user-interface design. *Computer*, 26(11):32–41, Nov. 1993.

[32] Node.js. http://nodejs.org/.

[33] PhantomCSS: CSS regression testing. https://github.com/Huddle/PhantomCSS.

[34] PhantomJS. http://phantomjs.org/.

[35] V. Quint and I. Vatton. Editing with style. In *Proceedings of the 2007 ACM symposium on Document engineering*, DocEng '07, pages 151–160, New York, NY, USA, 2007. ACM.

[36] D. Reed and J. Davies. The convergence of computer programming and graphic design. *J. Comput. Sci. Coll.*, 21(3):179–187, Feb. 2006.

[37] D. Roest. Automated Regression Testing of Ajax Web Applications. Master's thesis, Delft University of Technology, 2010.

[38] D. Roest, A. Mesbah, and A. v. Deursen. Regression testing ajax applications: Coping with dynamism. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 127–136, Washington, DC, USA, 2010. IEEE Computer Society.

[39] S. Roy Choudhary, H. Versee, and A. Orso. Webdiff: Automated identification of cross-browser issues in web applications. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[40] Sahi. http://sahi.co.in/.

[41] Selenium. http://docs.seleniumhq.org/.

[42] E. Soechting, K. Dobolyi, and W. Weimer. Syntactic regression testing for tree-structured output. In *In International Symposium on Web Systems Evolution*, 2009.

[43] G. Valiente. An efficient bottom-up distance between trees. In *Proceedings of the 8th International Symposium of String Processing and Information Retrieval*, pages 212–219. Press, 2001.

[44] W3C Recommendation of Cascading Style Sheets, Level 1, 1996. http://www.w3.org/TR/CSS1/.

[45] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: using gui screenshots for search and automation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, UIST '09, pages 183–192, 2009.