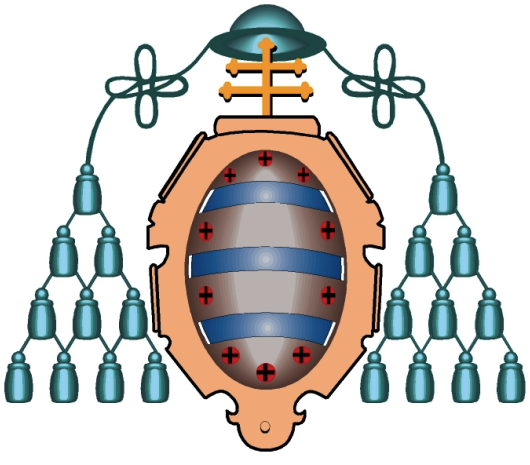
ALMcss: Separación de estructura y presentación en la web mediante posicionamiento avanzado en CSS

César Fernández Acebal



Universidad de Oviedo
Departamento de Informática

ALMcss: Separación de estructura y presentación en la web mediante posicionamiento avanzado en CSS

ALMcss: Separation between Structure and Presentation on the Web with CSS Advanced Layout

César Fernández Acebal

Universidad de Oviedo
Departamento de Informática

Tesis Doctoral presentada el 19 de marzo de 2010 para la obtención del título de Doctor por la Universidad de Oviedo (*Doctor Europeo*).

Dirigida por:

- Prof. Dr. Juan Manuel Cueva Lovelle (Universidad de Oviedo)
- Dr. Bert Bos (W3C)

Resumen

Desde el nacimiento de la web, y más concretamente con la aparición del primer navegador gráfico, los diseñadores web siempre han estado intentando mejorar la apariencia de las páginas, aun cuando eso llevase a ciertas perversiones en el uso de los estándares (o a no usarlos en absoluto). Las hojas de estilo (Cascading Style Sheets, o CSS) surgieron como un modo de devolver el HTML a su intención original, es decir, como lenguaje de representación de documentos estructurados. Hoy, trece años después de que apareciese la primera especificación de CSS definida por el W3C, las hojas de estilo se han convertido sin duda en una realidad ampliamente aceptada por la comunidad de diseño web, y todos los navegadores modernos las implementan razonablemente bien. Sin embargo, si bien es cierto que CSS ha logrado sacar fuera del HTML toda la información de estilo asociada a un documento, aún hay muchos sitios web que siguen utilizando tablas HTML para definir la maquetación. Y aun cuando se emplean diseños sin tablas y la maquetación se hace completamente con CSS, el marcado suele ser dependiente del aspecto final deseado.

El problema es que aunque CSS permite, en teoría, lograr cualquier diseño, lo cierto es que hay tareas que, por su cotidianidad, deberían ser triviales, y que sin embargo con las actuales capacidades de posicionamiento del lenguaje resultan demasiado complejas o incluso, en ocasiones, sencillamente no son posibles. Esta tesis sostiene que, por tanto, la prometida separación de presentación y contenido no se cumple, y que ello es debido a una ausencia de mejores herramientas de posicionamiento que hagan de CSS un verdadero lenguaje de maquetación.

Así pues, en la tesis se propone un nuevo mecanismo de posicionamiento para CSS, que ha sido desarrollado en el seno del Grupo de Trabajo de CSS del W3C (W3C CSS-WG), del que el autor de esta tesis y uno de sus directores son coautores: el CSS3 Template Layout Module. Tomando como base la teoría clásica de diseño gráfico de los *grid systems*, proporciona un marco para posicionar

elementos en cualquier lugar de la página, independientemente de cuál sea el lugar que ocupen en el documento de origen. Además, permite que la maquetación se defina ahora de manera explícita, a un nivel de abstracción mucho más alto que con las propiedades de posicionamiento de bajo nivel que proporciona el lenguaje en la actualidad, con las que la maquetación (*layout*) es sólo implícitamente definida por las complejas interacciones que tienen lugar entre dichas propiedades, aplicadas individualmente a cada elemento de la página. Como se demuestra en la tesis, esto proporciona varias ventajas sobre la forma actual de diseñar con CSS:

- Facilidad de uso
- Se provee un mecanismo de reordenación del contenido
- Los diseños son automatizables por una herramienta

A través de varios ejemplos y casos de estudio, la tesis demuestra cómo la solución propuesta permite una mayor separación entre la presentación y el contenido. O, más concretamente, entre la estructura del documento y su maquetación. Por último, se presenta un prototipo, llamado ALMcss (*Advanced Layout Module for Cascading Style Sheets*) que implementa dicha solución en los navegadores web actuales.

Abstract

From the beginning of the web and, more specifically, with the advent of the first graphical browser, web designers have been trying to improve the general appearance of their pages, even if it led to certain perversions in the use of standards —or to not using standards at all. Cascading Style Sheets (CSS) was born as a way to return HTML to its original inception (that is, as a language to represent structured documents), promoting a separation between presentation and content. Today, thirteen years after the W3C invented CSS, it has become a reality widely accepted by the web design community and well supported by all modern browsers. But, although it has achieved a great level of success in taking the stylistic information out of the HTML, there still are many sites that rely on HTML tables for their layout. And even when tableless, pure CSS layouts are used, the markup tends to depend, to a greater or lesser extent, on the final visual layout.

The problem is that, even though CSS makes it possible, in theory, to achieve almost any design, there are some common tasks which are difficult, if not impossible, to do with CSS. This thesis states that the promised separation between presentation and content is not currently possible in CSS, due to a lack of true layout mechanisms.

Therefore, this thesis proposes a new layout mechanism for CSS, which has been developed within the W3C Cascading Style Sheets Working Group (CSS-WG), coauthored by this author and one of his supervisors: the CSS3 Template Layout Module. It relies on the classical graphic design theory of grid systems, and provides a framework for arranging elements on the screen regardless of their position in the document source code. It allows the layout to be defined explicitly, at a high level of abstraction, whilst currently the layout is implicit, hidden in the complex interactions that happen among the low-level properties that are applied to every single element. As it is demonstrated in the body of the dissertation, it brings

several major advantages over the manner in which layouts are currently specified, to wit:

- Ease of use
- A content-reordering mechanism is provided
- Layouts can be automatised by a tool

This thesis demonstrates, through several case studies, that the proposed solution achieves a greater separation between presentation and content, or, more specifically, between the structure of the document and its visual layout on the screen when it is rendered. Finally, the thesis presents a prototype, named ALMcss (Advanced Layout Module for Cascading Style Sheets) that supports the new layout mechanism on current browsers.

Contents

References

Acknowledgments

If I could thank only one person in this thesis, that would be no doubt **Bert Bos**. Not only for his agreement to be one of the supervisors of this thesis —notwithstanding how much honoured I am by having as a supervisor the inventor of the technology to which this thesis is devoted—, but for his extraordinary personal support, both in the W3C CSS Working Group and during the three months I spent at W3C Office in Sophia Antipolis, in the summer of 2009 —how much I have missed our whiteboard sessions since them!—. I have no words to express my gratitude as he deserves.

Something similar happens with my other supervisor, **Juan Manuel Cueva**. Without his guidance, support, and encouragement, over the years, this thesis would not exist. I am sure that if finally becoming a doctor will take a load off my mind, he will also feel alleviated of not having to continually remember me: "César, you must do your thesis!"

I can not forget, either, **José Manuel Alonso**, former director of the W3C Spanish Office, who put me in contact with Bert Bos when I told him my initial thoughts about the possibility of contributing to improve the layout capabilities of Cascading Style Sheets, which in turn led me to join the W3C CSS Working Group and eventually to write this thesis, something that would have not been possible without his help.

My gratitude to **CTIC Foundation**[1] for having funded the first version of the prototype presented in this thesis, ALMcss, with the research project that also helped me to pay the travel and accommodation expenses of my first face to face (F2F) meetings as a member of the W3C CSS Working Group.

I am indebted to **María Rodríguez**, the first developer of the ALMcss prototype. This thesis exists thanks to the extraordinary work she performed during her research grant at CTIC Foundation, which I could never thank her enough. It is also compulsory to men-

1 www.fundacionctic.org

tion **Miguel García**, who continued María's work at CTIC Fundation, and the students **Enrique José Cabal** and **Pablo Abella**, who developed their undergraduate theses under my supervision and contributed to improve this thesis.

Thanks to all the members of the **W3C CSS Working Group**: it is a honour to work with so many talented people. I apologise for not having attending many teleconferences and some F2F meetings during the time I was writing this dissertation.

Special thanks to **Andy Clarke**, who honoured me allowing me to contribute to his splendid book, *Transcending CSS*, and who has dedicated to me kinder words than I deserve.

I will never forget the human warmth with which I was received by all the staff of W3C/ERCIM office at Sophia Antipolis, during my research stay there. I must explicitly cite **Pascale Peyrol**, **Caroline Baron**, and **Coralie Mercier**: their friendliness, *joie de vivre*, and their continuous jokes and laughs contributed to make my stay there much more enjoyable. Thank you, girls!

My most sincere gratitude to my fellows of **Laboratory of Object Oriented Technologies (Oviedo3)**, our research group at University of Oviedo, for all the support they have always given to me, specially during this academic year that I have been devoted almost full time to write this dissertation. I am privileged to work in such a fantastic atmosphere. I would like to extend this thankfulness to all the members of the Computer Science Department at University of Oviedo, both colleagues and administrative staff.

Also thanks to my students at University of Oviedo, who have understood that this year I have not been able to devote them so many time as I should.

The hardest sacrifice that I have had to make to finish this thesis has probably been not skiing this season, so some words of appreciation must go to **Guti**, **Kako**, **Luis Rovés**, **Julia**, **Elena**, and the rest of members of the **Alcoyano Ski Team**, as well as to **Andrade** and all my skiing buddies: you can not image how I have missed you! A special mention must be made to **Carlos Guerrero Castillo** (*Carolo*): he is not only one of the best ski coaches in the world, but

one of the most beautiful persons I have ever known, of whom I am lucky enough to count with his friendship.

To **Fermín**, **Marco**, and **Laureano**, classmates from primary to secondary school, and who still are my closest friends. My apologises for all the times I have not gone out to dinner with you because "I had to do my thesis".

To my Facebook friends, because during the last months of writing they have been almost my only contact with the outer world.

To my parents, for having educated me as they did. My mother has always been there, with her love and support. My parent taught me to prefer justice to peace, and he has always been a model of integrity and honesty to me. To my sister, hoping that this thesis serves her as a demonstration that everything can be achieved, and she becomes the professional dancer that she deserves to be. To all my family (cousins, uncles, grandparents...), with all my appreciation for the incredible good relationship that we maintain.

To my kitten **Lana**, who has also suffered this thesis, as her continuous "miaows" demanding my attention have remembered me during the last year that I have not played with her as I used to.

To **María**, my wife, my friend, my partner, with all my love, for the year I have spent sitting in front of my computer, nights and weekends included, and for her unconditional support. Thank you for being there the last sixteen years. Being always doing planes together about our future is what has given to me the strength enough to finish this thesis.

I would not like to finish these acknowledgments without remembering all my friends that, one way or another, have also suffered this thesis, which has been my recurrent and obsessive subject of conversation during many years. Although I can not mention every single person, this thesis is dedicated to all of you, since it is somewhat yours too.

1

Introduction

This initial chapter sets the hypothesis of this thesis: Cascading Style Sheets (CSS) is not a layout language. This is a bold affirmation that requires, of course, demonstration, and it is to what the first part of this thesis is committed. This chapter aims to answer the question of what this thesis is about and sets the foundation upon which the rest of the dissertation is based. Because of its introductory nature, it will inevitable make assumptions and refer to concepts that will be either demonstrated or thoroughly discussed in the body of the dissertation.

Finally, it outlines the structure of the whole dissertation, with a brief summary of every chapter.

Origin of This Thesis

I would like to start this thesis relating how it was conceived and the initial steps taken of what has been a long process until it has been eventually brought into existence in the form of this dissertation. Certainly, one does not wake up from bed one day and says: "*Hum, I'm going to do a Ph. D. Thesis on CSS!*". It is often only as a result of a longer mental process, mostly unconsciously, when one acquires the conviction that there is a field that is worth to be investigated.

Whereas this thesis has not been very different in that sense, there is, though, a precise instant when I started to think for the first time on the idea of doing my Ph. D. about this topic, despite it supposed a dramatic change from what had been my research until that moment, focused on things that had nothing to do with style sheets nor layout, like object orientation (Izquierdo, Acebal & Cueva, 2002), programming language design (Acebal, 2001), design patterns (Acebal, Izquierdo & Cueva, 2001), and that sort of things. Meantime I had also gained quite experience in web standards —specially CSS and HTML— and other related subjects such as usability and accessibility, but more as a teacher and an advanced user than as a researcher.

But then things suddenly changed. It was the year 2005 and I had to do a small web site for an introductory CSS course I was teaching at University of Oviedo since two years before (which I have continued imparting regularly since then, two or three times per academic year). For the second edition of that year I wanted to give the site the appearance of a magazine that had inspired me, a Spanish publication about interior decoration, called *Casaviva*. Among other design requirements, it should have the main navigation placed at the bottom of the page, imitating the main topics of the magazine cover. The final design of that web site, as it could be seen in 2005, is shown in figure 1.

As it can be seen in the sketch of the markup for the home page of that site shown in figure 2, the primary navigation, despite its desired position at the bottom of the page, is located at the top of the

Figure 1. Home page of the 2005 edition of the course on Cascading Style Sheets that I teach at University of Oviedo, inspired in the design of a magazine cover. The design of this web page led me to the hypothesis of this thesis: more advanced positioning and content reorder mechanisms need to be added to CSS to make possible a true separation between presentation and content on the web.

document, right after the header. That is its natural position when we concentrate on the logical order of the content rather than on the final presentation we want to give to it.

While doing that site, I noticed that we do not count with the needed tools in CSS to accomplish that sort of positioning. The layout is still feasible using just valid CSS and structural markup, as my own example is demonstrating: even in the browsers of four years ago —including Internet Explorer 6 with a couple of minor adjustments— the site was rendered correctly and it looked exactly like the modern capture of figure 1. The problem is that, although the

Figure 2. The primary navigation of the site, represented by the `ul` element with the identifier `menu`, is placed at the top of the HTML document, right after the header. That is its natural position when we think about the markup from a structural point of view, ignoring any consideration about which will be the visual design.

```
<div id="container">
  <div id="header">...</div>
  <ul id="menu">...</ul>
  <div id="main">
    <div id="content">
      <h1>Creación de sitios web mediante <strong>hojas de estilo</strong></h1>
      ...
    </div>
    <div id="footer">
      <p class="cesaracebal"><a href="http://cesaracebal.com"><img src="images/
barcode.png" alt="César Acebal"/></a></p>
    </div>
  </div>
</div>
```

menu appears at the bottom of the page, as it was intended, and it is even possible without introducing any change in the previous markup, it does not show the same behaviour as if it had been actually defined there in the HTML. This becomes obvious when we increase the font size or reduce the browser window. In both cases the menu overflows the footer, as it is depicted in figure 3.

Naturally, the question is why this is happening and if it could have been avoided using other CSS technique. As any experienced CSS user could guess just looking at that figure, the overflow occurs because I am using absolute positioning to take the menu out of its actual position in the markup (that is, near the top of the document, just after the name of the site) and display it at the bottom of the page. There is no other mechanism in CSS than absolute positioning that allows this type of severe changes between the position of an element in the markup and the place where it is visually arranged on the screen.

Figure 3. When the browser window is reduced the absolute positioned menu overflows the footer.

Of course, this could have much more easily achieved simply moving in the HTML the list that represents the menu to the footer:

```
<div id="footer">
   <ul id="menu"> ... </ul>
   ...
</div>
```

Not only would we obtain the same effect without the complexity of absolute positioning, but in that case the page would react as it is expected towards changes in the font size or the dimensions of the browser window. Because the menu is now actually contained in the footer, its parent element knows how to adapt itself to always fit its contents, without we have to concern about that (this is the default behaviour of HTML since the invention of the web). However, doing so would mean to alter the logical order of the markup, something undesirable in terms of the accessibility of the document (let us think, for example, how it would be interpreted by a screen reader or a Braille terminal). Moreover, this practice of introducing changes in the markup to obtain stylistic effects breaks the separation between structure and presentation. What would happen if later we wanted to redesign the site using a more common layout, with the navigation menu at the top of the page? Would do we then move it to the place where it should have been since the beginning if we had though in the content instead of the presentation?

Now, let us suppose that we had an imaginary CSS property that allowed us to place any element inside any other, no matter the order in which it actually appears in the HTML. Something like:

```
#menu {
  position: footer;
}
```

This thesis states that a true separation between presentation and content is not possible with current CSS layout mechanisms.

That simple. After all, have not we always been told that CSS allows the separation between presentation and content? This thesis states that such separation is not possible without more advanced mechanisms which let us put elements into any position of the page regardless of the place they occupy in the HTML.

Needless to say, things are not so easy as the *solution* barely sketched out in the previous code, of course, since there are many issues not yet addressed here. Specially, the order in which the so positioned element should be *inserted* into its destination. That is, how it interacts with the other elements that are actual children of the footer. In addition, while something like that would be a very flexible reorder mechanism, there is no evidence that it could serve as a general layout tool and not only for very concrete situations like the example presented here.

But the important thing is that I had got the conviction that there was a lack of powerful layout mechanisms in CSS. There was not the first time that I had been struggled with the problem of the order of the content versus the presentation, of course, but it was the first time that I thought of it in a conscious manner and the starting point for what has ended up being this thesis.

Joining the W3C CSS Working Group

Once it appeared clear to me that Cascading Style Sheets, in its current version, did not fulfil the requirements to accomplish the promised separation between presentation and content, and with just that vague idea in my head, I decided to talk with José Manuel Alonso, at that time director of the W3C Spanish Office, and express my thoughts to him. I had clear that there was still room for improvement in CSS with regards to its layout capabilities, but I was

not sure that it were a plausible topic for a doctoral thesis. I really needed the opinion of someone else. Ideally, someone with a solid background on CSS as to be able to confirm or reject the validity of my hypothesis, but who were also aware of what a Ph.D. Thesis must be. The person to whom he pointed me could not be more appropriate: Bert Bos. We had not only invented (with Håkon Wium Lie) the Cascading Style Sheets (Lie & Bos , 1996), but he was also in possession on his own Ph. D. degree (Bos, 1993).

He did not only not became surprised with my "proposal" (assuming that something outlined in a serviette can be so named), but it turned out that he was already working on the same problem (although with a much more elaborated solution: the Advanced Layout Module, an internal draft of the W3C CSS Working Group). The following is an excerpt of the very first electronic mail that he wrote to me, on February 8, 2005:

> *Something that the Device Independence WG is very interested in (and I as well) is to specify something like a "design grid" for a page or a site, using CSS. The idea is that a set of page templates is created for different types of devices, each of which describes the layout at a high level, e.g., 3 columns with a navigation bar at the top, two columns one of which has two rows, three columns of which one is twice as wide as the others, etc.*
>
> *It is possible to position elements of a page using the existing properties (margin, float, position, etc.), but such styles tend to be specific to a page and also quite hard to make, as soon as elements need to be shown out of order, e.g., to place a menu that is at the start of the file to the right of the main text on the screen.*
>
> *We (in particular Dave Raggett, Håkon Lie and myself) already wanted to develop a CSS module for such layout back in 1996, but at the time the browser makers weren't able (or willing) to implement it. But both graphic designers and content providers for mobile phones are asking for it now.*
>
> *My idea is that it is possible to define a CSS property, that defines for each element (typically for the root element) what the design grid for that element is. With some, hopefully simple, syntax, it describes the rows, columns, the spaces in between and the constraints on their sizes. All the other elements are then styled normally, but in addition, they are*

assigned to a slot in the grid: the menu goes in the menu slot, the logo goes in the logo slot, etc.

Thus, instead of working on some kind of academic solution that would have probably ended up gathering dust in a drawer, I decided to join Bert's efforts and start to work on the Advanced Layout Module. On the other hand, my University had recently became member of the W3C, so I talked to our W3C representative and, after a few months of bureaucracy, I was formally nominated to join the W3C CSS Working Group on 29th December 2005.

The Prototype

In parallel to my conversations with Bert Bos, and also thanks to the mediation of the W3C Spanish Office, I contacted CTIC Foundation[1], a regional organism of the Principality of Asturias, about the possibility of getting some funding research project to work on the ideas we have started to discuss. They were very interested in mobility aspects, and I was convinced that our solution could be also very helpful to people doing web sites for mobile devices, since it would make very easy to change from one layout to another without altering the HTML.

Thence, after a few meetings with the responsible people at CTIC Foundation, they agreed to give me a research project to develop a prototype which implemented the ideas sketched on the Advanced Layout Module. It was a very small project in terms of the money I received for it, but it helped me to fund my first travels to attend the W3C CSS Working Groups *face to face* meetings, and, which was much more important, the project included a research grant for a developer. As the main researcher of such project, I was the responsible to select the person for that position. After announcing the grant and doing quite a few interviews, I designated María Rodríguez as the grant holder, who started to work full time on CTIC Foundation offices in August 2005. She was therefore the first developer of the prototype presented in this thesis, which we

1 Center for the Development of Information and Communication Technologies in Asturias (CTIC Foundation), http://ctic.es/

named ALMcss (*Advanced Layout Module for Cascading Style Sheets*). By February 2006 we already had a first version of it that I was able to present at the W3C Technical Plennary of 2006 (Mandelieu la Napole, France), during a joint meeting between the CSS and Device Independent working groups.

Layout

The term *layout*, despite it is one of the essential concepts upon which this thesis is built, has already appeared several times so far in this chapter without any further explanation of its meaning. Although it will be defined as it deserves on *Chapter 3*, it is worth to provide here a brief definition of what I intend by layout in the context of this thesis.

In a rather informal definition, the layout of a document means in this dissertation the overall graphical structure of its elements when they are displayed on the screen, as opposed to other stylistic information such as fonts or colours. They are not completely separated, of course, because indenting or colouring a text influences what the user perceives as the visual structure of a page. But layout is usually situated at a higher abstraction level than those aforementioned presentational aspects.

Separation between Presentation and Content

This concept, expressed in any of its multiple forms ("separation between content and presentation", "separation between structure and presentation", "content vs. presentation" etcetera), is repeatedly cited as one of the main achievements of Cascading Style Sheets. However, it is ambiguous enough to mean different things for different authors. Thus, in the course of this research two distinct uses of that term have been identified in the literature:

Automatic layout

The separation between the presentation and the content of a document is sometimes a desired feature of certain publishing tools and platforms. The web itself is the best exponent of this architectural requirement (Jacobs & Walsh, 2004, §4.3):

> *The Web is a heterogeneous environment where a wide variety of agents provide access to content to users with a wide variety of capabilities. It is good practice for authors to create content that can reach the widest possible audience, including users with graphical desktop computers, handheld devices and mobile phones, users with disabilities who may require speech synthesisers, and devices not yet imagined. Furthermore, authors cannot predict in some cases how an agent will display or process their content. Experience shows that the separation of content, presentation, and interaction promotes the reuse and device-independence of content; this follows from the principle of orthogonal specifications.*

Or, as Hurst, Li and Marriot (2009) have pointed out:

> *In the last fifteen years there has been a resurgence of interest in automatic layout because of the World Wide Web (WWW) and variable data printing (VDP). This has resulted in a shift of focus from micro-typographic concerns such as line breaking to macro-typographic concerns such as page layout. One of the design goals of modern web document standards such as HTML and CSS has been to separate the document content from its presentation so as to allow the layout to adapt to different viewing devices and to different user requirements, such as for larger fonts. Further- more, dynamic content makes it impossible for the author to fully specify the final layout of a document.*

One of the meanings of separation between presentation and content refers to the ability of the system to automatically choose the most appropriate layout for a document based on its structure and on the concrete device where it is being rendered.

HTML is intended to be represented in a variety of devices, each with different constraints on screen size and resolution, available fonts, number of colours and so forth. Thus, one of the majors goals of HTML is to allow, even to impose, authors to concentrate on the content and structure of a document instead of its presentational aspects, which, in the absence of Cascading Style Sheets, are left to the user agent.

Therefore, in this context the duality between content and presentation is not referring to the author's capability to specify the design of a document after it has been created, but on the ability of the system to automatically choose the most appropriate layout for a document based on its structure.

Designer's viewpoint

Other interpretation of this phrase is often cited as a major goodness of Cascading Style Sheets and one of its design goals (Nielsen, 1997b):

> *Cascading style sheets (CSS) are an elegantly designed extension to the Web and one of the greatest hopes for recapturing the Web's ideal of* ***separation of presentation and content****.*

Nielsen's statement is alluding to the fundamental problem that is often implicit in most usages of this concept: although the web had been born as a medium independent of any specific hardware device and software platform, and HTML had been conceived as a language to represent just the structure and content of a document, designers soon started to deviate from its inception, and began to use it more as a presentational language. According to this interpretation, Cascading Style Sheets were developed to allow authors to work on the structure and content of a document without worrying about its visual representation. It would be later, once the document were finished, when they could apply whatever style to it, and even change from one design to another without modifying the HTML document.

The last meaning of separation between content and presentation is the approach with which this thesis is concerned (although to refute it: as it will be demonstrated in subsequent chapters, such degree of separation is not yet possible on the web, due to the lack of true layout mechanisms in CSS).

Problem Statement

The opening example of this chapter already provided some hints of the problem addressed by this theses, namely, its inability to carry out some layouts without altering the logical order of the source code or adding extraneous markup, something that breaks the promised separation between presentation and content. This section introduces it in more detail, although the complete explanation will be deferred until *a later chapter*.

A Historical Perspective

From the beginning of the web and, more specifically, with the advent of the first graphical browser, web designers have been trying to improve the general appearance of their pages, even if it led to certain perversions in the use of standards —or to not using standards at all.

Several of such tricks are well-known to any experienced designer: transparent images, using tables for composition or *esoteric*, non semantic uses of some HTML elements are just a few of them. Let us think a moment about how people used those old techniques and we will realise that many of them were aimed at changing the **layout** of the document. Indeed, one of the first things that designers missed in the web was the ability to specify the overall visual structure of a page with the same easy and level of control that they were used to in traditional, printed media.

These kinds of complex layouts were impossible in the first days of the web. After all, HTML was not designed for that, but as a way to represent the *content* and *structure* of a document. But the addition of the `table` element to HTML 3.2, in 1997, opened up a new world of possibilities for designers, who soon began to employ it not to represent tabular data, but to specify the layout in terms of rows and columns (that is, as a *grid*). We all know what followed: designs in which content was insignificant compared to the amount of nested tables. It is worth recalling the problems of those deeply nested, table-based layouts:

- They **mix content with presentation**, sometimes to such extremes that the task of maintaining the site (adding content to a page or redesigning) becomes a nightmare.
- They present well-known **accessibility** issues, because it is difficult for screen readers and mobile devices to interpret them right.

In an attempt to turn the situation around and return HTML to its original conception as a language to represent the logical structure of web documents, in 1996 the W3C developed *Cascading Style Sheets* (Lie & Bos , 1996). All stylistic information should now be taken out of the HTML and left to the style sheet.

While CSS has achieved a great amount of success in removing, for instance, `font` tags from the markup, there are many sites that still rely on HTML tables for specifying their layout. This is very common in web sites that are redesigned to be standards compliant. Although they use valid (X)HTML and CSS for most of the stylistic information of the page, many of them continue specifying their layout with tables. This is what has been called *hybrid* layouts, as opposed to *pure* CSS layouts (Zeldman, 2003).

Table-based layouts

Despite their aforementioned well-known problems, tables are still a common way of laying out web pages. *Why?*

We should not oversimplify and blame designers saying that many of them are so used to those old techniques that they have not been able to adapt themselves to web standards. Although there are, without doubt, people that effectively have not made an effort to understand CSS, the truth is that many professional designers have never felt comfortable laying out their sites in terms of floats and absolute positioning (and they probably do not lack reasons). No doubt, inconsistencies between browser implementation of CSS are also responsible for this attitude.

The problem is that, although CSS is said to provide great flexibility to layout a page and makes possible, in theory, to achieve almost any design, there are some common tasks which are difficult, if not impossible, to do with CSS —let us think, for example, how many articles, tutorials and blog entries have been written about getting equal height multi-column layouts. As Holzschlag (2005a) has pointed out:

> *What we're just beginning to understand —particularly those of us who come to CSS layouts after years of working with tables— is that the visual model for CSS is far more conducive to breaking out of the grid and designing for discrete, semantic elements. Perfect, no, for despite the gains made possible by CSS, we lose things too. Stretching columns is a decidedly problematic issue in CSS design, and cell spacing is too.*

Thesis Structure

Being this a thesis that falls into several fields, it is compulsory to provide some background information before to examine the problem in detail. That is what chapters 2 and 3 do. The former presents the basic concept of separation between structure and content, and reviews the related literature. The latter do the same for layout and, more specifically, the fundamental graphic design theory of grid systems, which has inspired the solution presented in this thesis.

After that, some layout languages are reviewed in chapter 4, since they are related, to a greater or a lesser extent, to the proposed solution.

Chapters 5, 6, and 7 aim to demonstrate the hypothesis of this thesis, namely, that CSS is not a layout language and, as a consequence, the promised separation between presentation and content is not currently possible. This part of the dissertation begins reviewing the basis of the Cascading Style Sheets box and visual formatting models, then analyses the state of the art of advanced CSS techniques that are being used for layout, and, finally, provides a few case studies, of different levels of complexity, to proof the hypothesis of the thesis.

Chapter 8 summarises and discusses the issues of CSS that have been identified in the previous chapters, setting the problem statement of the thesis in detail.

Then, it starts the part of the dissertation where a solution to this problem is proposed (Chapter 9). It is demonstrated (Chapter 10) with the same case studies that were previously reviewed on Chapter 7, which are now done with the extension to CSS presented in the previous chapter. Finally, an implementation of the proposed

solution that works in current web browsers is thoroughly described (Chapter 11), and a prototype of a visual layout generator that uses the proposed template layout mechanism is presented on Chapter 12.

Last chapter summarises the major contributions of this thesis and where there is room for further research on the subject of layout and Cascading Style Sheets.

After this introduction, a brief description of each chapter is provided below.

Chapter 1. Introduction

The aim of this chapter is to set the problem statement of this thesis. Of course, this has necessarily to be done at a high abstraction level, since the details can only be totally understood once the layout capabilities of Cascading Style Sheets have been discussed in Chapters 5 and 6.

Chapter 2. Separation between Presentation and Content

This chapter describes the concept of separation between presentation and content, a common principle of structured documents and style sheets languages. It reviews the related literature to later concentrate on the specific case of World Wide Web.

Chapter 3. What Layout Is

The third chapter defines what layout is in the context of this thesis and provides a background on the theoretical aspects of graphical design which are relevant for this work. It briefly introduces grid systems, a classical concept of graphic design widely used in printed media that only in recent years has commenced to gain popularity among the community of web designers.

Chapter 4. Layout Languages

Prior to study how Cascading Style Sheets allow us to specify the layout of web documents, this chapter will review how other languages address the same problem of layout. The languages reviewed in this chapter are not intended as alternatives to the solution

presented in this thesis, but as inspiration for the requirements of such solution.

Chapter 5. CSS Box and Visual Formatting Models

This chapter reviews how CSS currently works with regards to layout. To do so, the box model and visual formatting model of Cascading Style Sheets are described, as well as the related properties which deal with box dimensions and positioning schemes. I am aware that this can be boring for some people who already have a solid understanding of Cascading Style Sheets. It does not escape me neither that a Ph. D. Thesis is not supposed to be a book on CSS —not even an advanced one. However, I have finally decided to include this review of the way of working of CSS for the sake of completeness. First, this thesis is built on the assumption that the tools provided by CSS for layout purposes (namely, floats and absolute positioning) are far from being ideal. To demonstrate such affirmation I need to make sure that all the intricacies of such techniques are shown up. Secondly, I did not want that a not so well versed on CSS reader of this thesis had to make constant references to the specification to be able to understand the examples and case studies provided to demonstrate the inability or complexity of CSS to carry out certain layouts.

Chapter 6. CSS Layout Techniques

Once described how CSS works, some more advanced techniques are presented in this chapter. They use the properties seen before in more complex and imaginative ways to obtain some layout effects which are not evident, like getting any number of columns in any order, or equal height columns. This chapter thus represents the state of the art of this thesis with respect to the capabilities of CSS as a layout language, setting the foundation for the problem statement of Chapter 8.

Chapter 7. Case Studies

Once the current layout mechanisms of Cascading Style Sheets have been discussed in the two previous chapters, it is time to put

them in practice and experiment with some layouts to see how they can be created with CSS. This chapter dissects the construction process of a few case studies to extract the conclusions that are going to be discussed in the next chapter, the problem statement.

Chapter 8. The Problem of Separation between Structure and Layout

Although the main problem addressed by this thesis has already been outlined in this introduction, this chapter explains it in more detail, summarising the results of the previous experimental chapter and discussing the problem statement of this thesis, namely, the lack of proper layout mechanisms in Cascading Style Sheets, which prevents a true separation between presentation and content.

Chapter 9. Proposed Solution: the CSS3 Template Layout Module

A concrete solution to the problem of layout on the web is presented in this chapter, in the form of an extension to CSS to give it true layout mechanisms that do not rely on floats and absolute positioning and allow instead to define the overall layout of a document in an explicit way. The proposed solution has been developed inside the W3C CSS Working Group, and is a Working Draft of the future specification of Cascading Style Sheets. Some extensions that do not form part yet of the working draft are also proposed.

Chapter 10. Demonstration: Case Studies Revisited

To demonstrate the goodness of the solution to resolve the problem of the lack of true layout mechanisms in Cascading Style Sheets, this chapter revisits the case studies that were presented in Chapter 7 and explains how they could have been achieved using the Template Layout Module.

Chapter 11. ALMcss: A JavaScript Implementation of the Template Layout Module

In addition to the theoretical solution proposed in Chapter 9, and to the demonstration of how it could have been used in practice to

carry out some designs that are either impossible or very complex to do with CSS, it is still needed to prove that such solution can be actually implemented. Therefore, this chapter presents the prototype that has been developed as a part of this research: a JavaScript browser plugin that implements the Template Layout Module in current browsers.

Chapter 12. A Visual Layout Generator

Another contribution of the proposed solution is that it should make easier for visual editing tools to generate layouts that it is today. Although WYSIWYG applications have improved a lot during the last years, and they are now able to produce pure CSS layouts, the code they generate is still inelegant and verbose, and they compromise the logical order of the content. Furthermore, they usually rely on absolute positioning for layout, which leads to inflexible designs that does not adapt themselves well to different screen resolutions and mobile devices. This chapter presents a prototype that is able to generate templates for the proposed solution, to proof that it can be easily automatised, and to make easy for final users to use the module without having to know its concrete syntax.

Chapter 13. Conclusions and Further Research

The ending chapter of this dissertation summarises the main contributions of this thesis: first, it has demonstrated the unsuitableness of Cascading Style Sheets to be used for layout, and, for that very reason, that a true separation between presentation and content is not currently feasible on the web; secondly, the requirements that an extension to CSS should fulfil to be considered a solution to that problem have been identified; finally, a concrete solution, along with the demonstration of its viability, and an implementation that serves as a proof of concept are also provided. Furthermore, the chapter analyses where there is room for improvement in the subject of layout on the web in general, pointing out some lines of future research in this field.

2

Separation between Presentation and Content

Although the idea of separation between presentation and content is well-known, and, specially in the case of the web and Cascading Style Sheets in particular is repeatedly cited, it is necessary to explain what it means and how it is connected to this thesis.

Therefore, this chapter first introduces such a concept, briefly reviewing the related literature, and then moves into the World Wide Web concrete issues, where the problem of separation between content and presentation is put into its historical context, providing some background on how this idea was first misunderstood and then perverted.

Introduction

The concept of separation between presentation and content is neither new nor exclusive to the web or CSS. Nor even to electronic publishing. Although there is much written on this subject in the fields of *document engineering* and *document formatting*, as Fleishman pointed out (1999), the idea of style sheets, based on the concept of "define once in a central location and apply many times throughout a document", can be even found in the pre-desktop-publishing days:

> *Back then, when designers needed to do elaborate formatting for body copy, headings, and so on, they would create a list of specifications —on a typewriter, even— and assign a number to each style. When a typesetter saw a circled 1, for instance, he or she referred to style 1 on the style sheet and carried out its specifications.*

It was, however, with the lowering of cost in microcomputers and imaging devices, when the two areas of typesetting and publishing, and computer science, were brought together, not only for the benefit of the brute-force computer power, but also thanks to the application of computer science concepts to the field of electronic publishing (Brailsford, 1988). According to this author, there are analogies between typesetting and compilers, in the sense that, by having a machine-independent intermediate code, a compiler becomes usable over a wider range of computers; similarly, "the idea of a document being an abstract concept, *which may or may not be related to its final concrete appearance on the printed page*", such as happens with the Standard Generalized Markup Language (SGML), makes possible to process it "in ways which go far beyond text preparation". Interestingly, the same analogy between compilers and electronic publishing was also simultaneously done by Chen and Harrison (1988), when they say that "traditional document development systems like the Troff family, Scribe, TeX and SGML ... are largely noninteractive language compilers".

But, what does *separation between presentation and content* mean? Essentially, it refers to the fact that an author can (and *should*) focus

only on the content of a document (that is, the writing, figures, and tables, in the case of a printed document, like for example this dissertation; or any other content, like video or audio, in multimedia documents), which will be later formatted to give it the desired appearance (page size, font type and size, margins, colours, etcetera). This is frequently cited as one of the essential features of separation of presentation and content in general (Clark, 2008), and of *descriptive* markup languages in particular (Coombs, Renear & DeRose, 1987, p. 943).

However, the above definition of is too simplistic. Even after completely separating the presentation from it, any document is more than the content itself. Thus, the content in inextricably bound to the *structure* of the document. Hence the notion of *structured documents*, essential to style sheets: without structure, no style can be later applied to a document.

As for the style, van Ossenbruggen and Hardman (2002) make a similar distinction: they argue than the "structure of the presentation" (what in this thesis terminology is called *layout*) is a third essential ingredient in the separation of concerns of a document:

> *The simple separation of content and style as described above suffices only when the presentation structure is similar to the content structure in the underlying XML. If this is not the case, then a transformation step, such as enabled by XSLT, is needed to convert the content structure to the desired presentation structure. For example, the lexical order in a source HTML document might need to be transformed to the order that is most appropriate in the text-flow of the target HTML presentation.*

Although their research is more concerned with multimedia, the quotation above can be extrapolated to HTML/CSS, and is very relevant to the problem tackled by this thesis. Based on their reasoning, they establish several dependencies among the three identified ingredients of a document, which are depicted in figure 1.

It is also worth to mention the distinction that van Ossenbruggen (2001, pp. 11–12) establish between **layout-driven versus content-driven applications**:

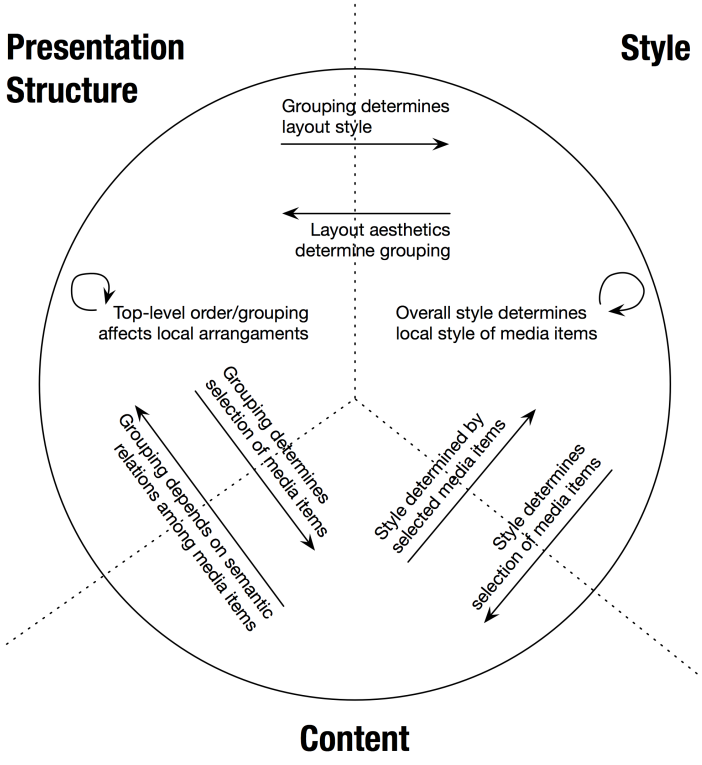


Figure 1. Dependencies between content, presentation structure, and style, as shown by van Ossenbruggen and Hardman (2002).

In layout-driven applications, the layout and content of a document are tightly coupled, and there is generally no need to produce multiple versions with alternative layouts. Typical examples of layout-driven applications include the cover page of a glossy magazine or advertisements with a large amount of graphical material. Because of the tight integration of content and layout, these documents can be effectively authored directly in terms of the final presentation.

In contrast, content-driven applications focus on the content, which often needs to be presented in several ways, for example by using different layouts. For content-driven applications it is useful to separate content from presentation information, because this separation allows reuse of the same content in alternative presentations.

Structured Documents

Essential to the separation between presentation of content is the notion of *structured documents*, in which "the relationships between components are based on the document's logical structure and not the physical appearance of the components on the page" (Furuta, Quint & André, 1988). These documents are usually represented by *generic markup*, a concept that was first originated by IBM Generalized Markup Language (GML) and then popularised by Scribe (Reid , 1981). One of the postulates of GML, according to Goldfarb (1981) was:

> *Markup should describe a document's structure and other attributes, rather than specify processing to be performed on it, as descriptive markup need be done only once and will suffice for all future processing.*

Furuta, Quint & André (1988) distinguish three representations of a document for being processed, to wit:

- The document model representation
- The output model representation
- The display representation

The document model represents the *logical structure* of the document, which will be later transformed into the output model through the *formatting* process. The output model represents the physical appearance of the document. This output model is finally mapped to the specific display representation, depending on the particular medium in which it is going to be rendered.

Separation between Presentation and Content on the Web

Nielsen (1997b), one year after the first version of CSS had been released (Lie & Bos , 1996), already stated that:

> *Cascading style sheets (CSS) are an elegantly designed extension to the Web and one of the greatest hopes for recapturing the Web's ideal of separation of presentation and content.*

Nielsen does not only affirm that one of the basic principles of the web is the separation of presentation and content, but he is also implying that that ideal had been lost at some point along the path and it was needed to recover it. This section recreates that journey, from the architectural requirements of the web itself to the invention of Cascading Style Sheets, focusing on the mentioned goal of separation between presentation and content, which is put here in its historical context, providing thence a foundation for what will come later on *Chapter 8*: a compilation of my own conclusions about the specific problem tackled by this thesis, namely, the lack of advanced layout mechanisms in CSS which prevent this separation of concerns to be true.

The Web as a Universal Platform

The World Wide Web (WWW) was conceived as a universal platform: independent of any specific hardware device, software platform, language, culture, or disability (Berners-Lee, 2007). Let us focus just on the first requirement of those enumerated in the previous sentence: the *device independency*. Its obvious meaning is that a user should be able to access to any web site from any device, regardless of its concrete features such as installed fonts, colour depth, or screen resolution. But it also has implications on the way that web sites are designed. Specifically, it means that authors can no longer think in the WYSIWYG way they were used to in printed media and traditional publishing tools. Whereas a magazine designer has total control over every aspect on the magazine page, such as page size and colours, the same is simply impossible, by definition, on the web. We can not pretend that every user sees exactly the same page, for the simply reason that we do not know the size of his browser window (not to mention other devices such as text browsers, braille devices or voice synthesisers). Thus, instead of attempting to recreate exactly the same visual aspect for every user, designers must specify the web pages in terms that allow browsers to optimise them based on the individual circumstances of every user (Nielsen, 2000, p. 28).

WYSIWYG and pixel level control are, by definition, impossible on the Web.

HTML as a Language for Representing Structured Documents

To accomplish the requirements excerpted in the previous section, Berners-Lee (1993) created the HyperText Markup Language (HTML). It was strongly based on SGML, an ISO standard for defining markup languages for documents. Despite HTML, strictly speaking, is not an instance of SGML, it has inherited many of its postulates, one of which is that *markup should describe the structure of a document, rather than any of its presentational aspects*. HTML is therefore a document format that is device-independent: it can be rendered into many different devices, such as printers, screens, braille printers and text synthesisers (Berners-Lee, 1991):

> *It is required that HTML be a common language between all platforms. This implies no device-specific markup, or anything which requires control over fonts or colors, for example. This is in keeping with the SGML ideal.*

Lie (2005) has studied and classified various document formats based on their ladder of abstraction. The results of such classification are summarised in the table of figure 2. Despite the high level of abstraction that the author concedes to HTML, which makes it appropriate for being presented in very different ways, Lie recognised that his rating of HTML "is based on a best-case scenario where the author makes use of semantic elements and does not alter the reading order of elements by using features such as positioning or tables. *It may be argued that most HTML documents do not follow these conventions*" (own emphasis).

But the problem is not whether the creator of an HTML document uses Cascading Style Sheets positioning or other layout mechanisms to alter the linear order of the content when it is visually rendered: as long as all of those transformation take place in the style sheet, there is nothing bad in allowing the author to lay out the content so that it conveys more effectively its message (that is supposed to be the graphic designer's mission, after all). In that sense, Lie's words are confusing when he says: "… and does not alter the reading order of elements by using features such as positioning or tables", because when he uses the term "positioning" it is not clear

	GIF, PNG	private XML vocabulary	PDF	XSL-FO	HTML	MathML
application-specific semantics?	no	no	no	no	no	yes
device-independent?	no	no	no	no	yes	yes
roles known?	no	no	no	no	yes	yes
text in logical order?	unknown	unknown	no	yes	yes	yes
reflow possible?	no	unknown	no	yes	yes	yes
scalable?	no	unknown	yes	yes	yes	yes
text machine-readable?	no	yes	yes	yes	yes	yes
text human-readable?	yes	yes	yes	yes	yes	yes

Figure 2. A classification of various document formats with respect to the ladder of abstraction, as it appears in Lie (2005, p. 42).

whether he is referring to CSS positioning mechanisms, which do not alter the reading order of the content in any way.

The problem comes when those mechanisms are not powerful enough to let the author present the content in the desired position and he is therefore doomed to add superfluous markup, move an element to another position in the HTML source code, or use tables for layout (which is a particular case and has the effects of the two previous methods). Any of these actions breaks the separation between presentation and content, because we are binding the markup to an specific layout and, when the latter changes, the former must be modified too.

Old School Tricks

While the web was composed only of text pages that appeared in black and white (or black and green) on the monochrome monitors of that time, HTML remained as the structural language that it was supposed to be. But, once the first graphical browser, Mosaic[1], appeared in 1993, companies began to discover the potential of the web for commercial purposes and they soon started to demand more and more appealing web sites. As a result of this interest, web designers commenced to use HTML in a way for which it had not been conceived. This problem has been acknowledged by many authors. Thus, Korpela (1998) stated:

> *In traditional publishing, graphic designers and layout artists consider the specific features of the presentation medium, including the paper size and quality, the color palette, and so on. It has been very difficult to switch from this approach to a simpler one, in which the author provides the content and specifies the logical structure, leaving the presentation to user agents.*

Some of those tricks include (Lie & Bos, 2005, p. 6):

Incorrect use of HTML elements

For example, they soon discovered that a `li` element without its counterpart, `ul`, was rendered by most browsers without the characteristic bullet of unordered lists, but preserving the indentation, so it became a common way to get indented text on web pages, even though it broke the grammar of HTML.

Another example is the use of header elements just for the purpose of getting bold and bigger text to highlight some phrases. Although this does not invalidate the document, this sort of non semantic use of elements in general and headers in particular presents many accessibility problems.

1 http://www.ncsa.illinois.edu/Projects/mosaic.html

Figure 3. The Hook Mitchell website (www.hookmitchell.com). As Zeldman says (Zeldman & Marcotte, 2010), "a compelling design powered by contorted code". It is entirely made of images, instead of text, which in addition are enclosed in tables.

Proprietary HTML extensions

Once it became clear that authors were demanding more and more control over the presentation of web pages, browser vendors started to introduce presentational elements that deviated HTML from its conception. In addition, since those elements did not form part of the standard, they only worked in each specific browser. This was known as the *Browser War*, referring to the competition between the two major web browsers in the early days of the web: Netscape and Microsoft Internet Explorer. Some of the invented elements were eventually became part of the HTML specification (as happened with `FONT` or `TABLE`); others, like `MARQUEE` or `BLINK` never did.

Using images instead of text

Another way to overcome what designers perceived as a limitation of HTML (that is, its inability to define the presentation of a document) was to convert text into images. By using an image, designers were able to include any font and fully control other features, such as spacing, alignment, or any other presentational aspects. Of course, this goes against the nature of the HTML and the web itself,

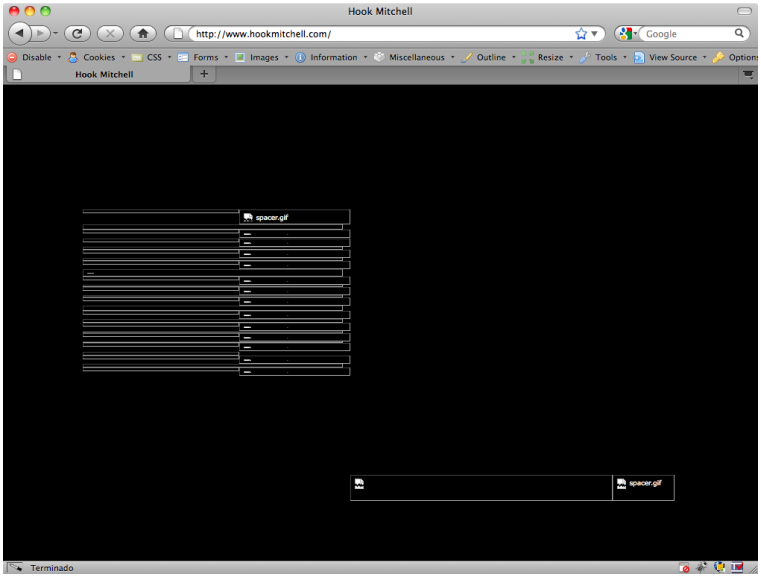


Figure 4. The same film site, this time with the images disabled.

and makes the page *inaccessible* to other devices and search engines, as it can be seen in the website of the film Hook Mitchell, cited by Zeldman (Zeldman & Marcotte, 2010, pp. 38–41), shown in figure 3, and in figure 4 with images disabled.

Using tables for layout

This method, which has already been mentioned in the introduction of this dissertation (see p. 13), consists on using tables not for representing *tabular* data, but as a way of arranging the elements on the page, that is, for *layout*. While some of the other tricks are something of the past, the use of HTML tables for layout has remained until today and they are still prevailing[1]. This issue is so relevant to the problem tackled by this thesis that it will be described in more detail in a following section.

Writing a program instead of using HTML

Another technique that allows designers to have full control over the final appearance of their pages is writing some program that is executed by the browser, instead of using the lingua franca of the

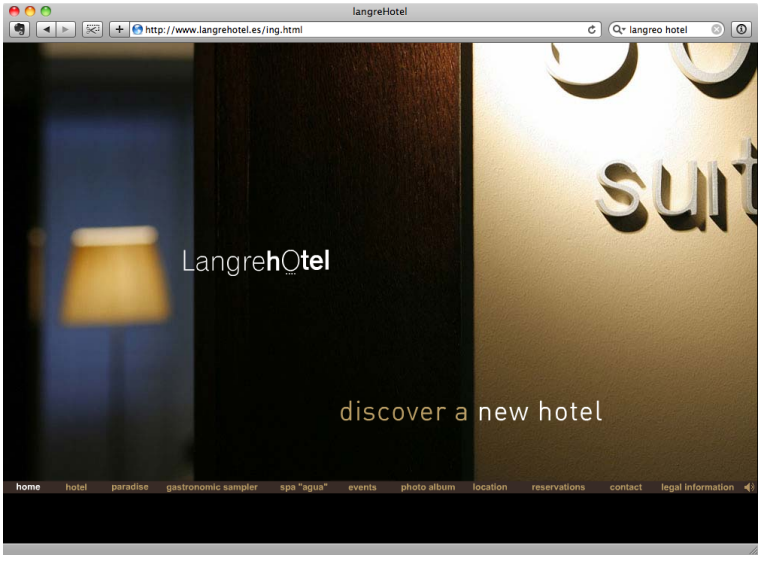


Figure 5. LangrehOtel website (www.langrehotel.es), entirely made in Flash.

web that is HTML and the related standards (CSS for presentation and JavaScript for behaviour). Examples of this technique are Java Applets and, more recently, Flash. It shares most of the accessibility problems of images, as it can be seen in figure 5, which shows the website of LangrehOtel, a modern and well designed hotel located at La Felguera (Asturias, Spain), but which website, entirely made in Flash with no reason for it (exactly the same site, but *better*, could have been done with standards), is totally inaccessible, as shown in figure 6.

1 In a survey of over 21,500 web pages, Levering and Cutler (2006) found that only about 15.1% (CI=0.7%) of documents did not use a table at all. Although this fact alone may not be very significative (tables might be being used for representing *tabular* data), they also found that, of the documents that did use tables, they averaged maximum table depths of 2.95 (CI=.04), reaching up to 8 tables at the high end. As the authors say, this "indicates that nesting of tables to achieve the desired layout is a norm". A subsequent study made in 2007 by Levering for the book *Website Optimimization* (King, 2008) found that, despite the widespread adoption of CSS, 62.6% of web pages still use tables for layout —and 32.8% use the `font` tag for inline style—, although the average table depth decreased from 2.95 to 1.47 with respect to the 2006 survey (Website Optimization, 2008).

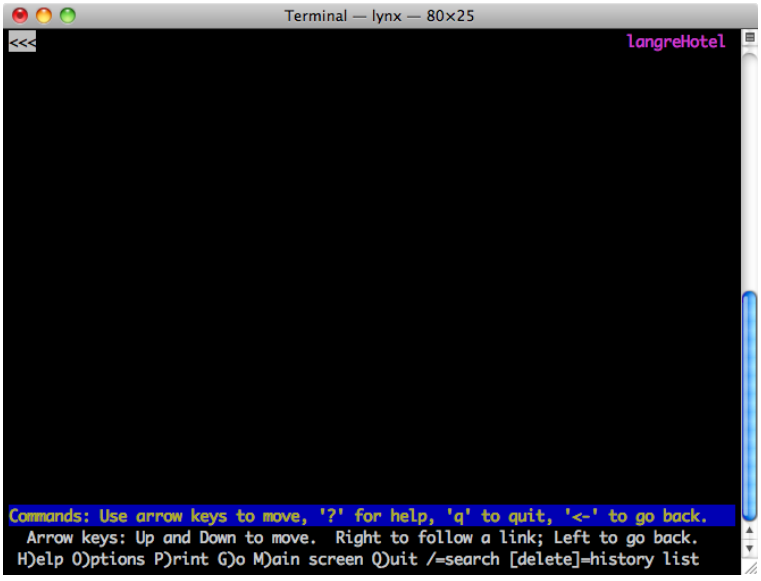


Figure 6. LangrehOtel website, as seen in the Lynx text browser: a blank screen where only the title is visible, making this site totally inaccessible to text browsers, braille devices, voice synthesisers, search engines, or any other device that does not support Flash (many mobile phones, iPod, iPad…).

The problem with the above methods is that they abandon the idea of HTML as a language for representing structured documents and try to use if for things for which it was never conceived, specifying *how* a document should look like, instead of defining its logical structure. In addition to the aforementioned accessibility problems, there are *usability* issues, too: users have accustomed to some features of HTML documents when they are rendered by a web browser, and anything that deviates from their expectations may cause confusion. One example of this are the custom scrollbars, many of them programmed in Flash, which can be found in many sites, instead of the default scrollbar automatically provided by the web browser through the native window system of the platform. Not only must the user to learn how to use them, but sometimes users not even *notice* them (Acebal, Cueva & Izquierdo, 2003; Nielsen, 2005). Other issues are related with some browser features that do not work (or, at best, behave *differently*), such as the back button, the ability to increase or reduce the font size, the "find in this page" option (Nielsen, 2000b), the secondary button of the mouse (for example, to open a link in a new tab), etcetera.

Presentational Elements

As it has been noted above, some of the presentational elements that had been invented by browser vendors, to respond to the demand of web designers to control the appearance of web pages, were eventually incorporated to the W3C HTML specification. Thus, `CENTER` element, for instance, was added to HTML 3.2 specification (Raggett, 1997). But the most (in)famous element of those officially added to HTML specification was `FONT`. With it, designers could change the colour and size of the font, and, in some browsers, also the font face, through the `face` attribute, which was finally also added to HTML 4.0 (Raggett, Le Hors & Jacobs, 1997; Raggett, Le Hors & Jacobs, 1998). By including this element in the specification, HTML lost its meaning as a language for representing structured documents, and soon web pages started to contain more presentational code than the information and structure actually contained in them. An example of this —as well as of the use of tables for layout and other common ills of the web a few years ago— can be seen in figure 7, which depicts a piece of content of the Yahoo! home page in 2003[1].

Use of Tables for Layout

Tables had been absent from the first versions of HTML and were added to HTML 3.2 (Raggett, 1997) as a subset of the table model previously proposed by Ragget (Raggett, 1996). Even the specification itself states that tables "can be used to markup tabular material *or for layout purposes*" (emphasis is mine), although it is true that it continues saying: "Note that the latter role typically causes problems when rending to speech or to text only user agents". This was fixed in HTML 4.0 (Raggett, Le Hors & Jacobs, 1997, §11.1), which now specifies clearly that they are intended to arrange *tabular* data and must not be used for *layout* purposes:

> *Tables should not be used purely as a means to layout document content as this may present problems when rendering to non-visual media. Additionally, when used with graphics, these tables may force users to scroll*

1 Accessible in web.archive.org/web/20030127104212/www.yahoo.com/

> *horizontally to view a table designed on a system with a larger display. To minimize these problems, authors should use style sheets to control layout rather than tables.*

The reality, though, is that tables had already began to be used almost exclusively for layout, once browser vendors implemented them first as a proprietary extension and designers rapidly discovered their potential. Prior to them, it was totally impossible, for example, to achieve something apparently as simple as two columns (remember that Cascading Style Sheets had not yet been invented). The problem is that, whereas most of the other reviewed practices remain only as vestiges of that pre-CSS era, and the use of `font`, for instance, is now marginal, *tables are still a widespread layout technique*.

Why are Tables Bad for Layout?

Much has been written about the problems of using tables for layout (Lie & Bos, 2005, p. 8; Merikallio & Pratt, 2003; Chromatic, 2008; ...). This section summarises and briefly discussed the main ones.

Accessibility Issues

Probably the most cited reason against the use of tables for layout is related to their *accessibility* flaws. In effect, tables may run into accessibility issues, but it is also true that this is the most easily solvable of their problems. This issue has to do with the correct order of the content from a logical point of view, and with the general recommendation of Web Content Accessibility Guidelines 1.0 (Chisholm, Vanderheiden & Jacobs, 1999, §3.3) of using "style sheets to control layout and presentation". More specifically, Techniques for Web Content Accessibility Guidelines 1.0 states the following (Chisholm, Vanderheiden & Jacobs, 2000, §3.3):

> *Do not use tables for layout unless the table makes sense when linearized. Otherwise, if the table does not make sense, provide an alternative equivalent (which may be a linearized version).*

As for the **linearisation** term that is mentioned in the quote above, it is further explained in the HTML Techniques for Web Content Accessibility Guidelines 1.0 (Chisholm, Vanderheiden & Jacobs, 2000, §5.3), and it basically consists on making sure that if the table

```
</font></td><td align=right nowrap><font face=arial size=-1><b><a href=r/
xy>More Yahoo!...</a></b></font></td></tr>
</table>
<small><small><br></small></small>
<center>
<script language=javascript>
document.write('<div id=mdiv');
document.write('><table border=0 cellspacing=0 cellpadding=1
bgcolor=498eb6><tr><td><table border=0 width=100% cellspacing=0 cellpadding=0
bgcolor=white><tr><td align=right valign=top bgcolor=ebebeb nowrap width=160
rowspan=3><table border=0 cellspacing=0 cellpadding=0><tr><td align=right
valign=top nowrap><a href='+jp(1,'img1',img1u)+'><img src='+img1+'
width='+imgw+' height='+imgh+' border=0></a></td></tr>');
document.write('<tr><td align=center height=28 nowrap><font face=verdana
size=-2><a href="'+jp(4,'txt1',txt1u)+'">'+txt1+'</a><br>');
document.write('</td></tr></table></td><td valign=top><a href=s/55303><img
src=http://us.i1.yimg.com/us.yimg.com/i/mntl/spo/03q1/sb_hdr_3.gif alt="Yahoo!
Sports" width=273 height=43 border=0></a></td></tr><tr><td nowrap align=center
valign=top><table cellspacing=0 cellpadding=1 border=0 bgcolor=f5af2a
width=97%><tr><td valign=top><table cellspacing=0 cellpadding=3 border=0
width=100% bgcolor=white><tr><td align=center nowrap><font face=verdana
size=-2><b><font color=d4012e>Final Score:</font> <a href=s/55404>Tampa Bay</a>
48</b> - <a href=s/55405>Oakland</a>
21</font></td></tr></table></td></tr></table></td></tr><tr><td align=center
valign=top><table border=0 cellspacing=0 cellpadding=0 width=100%><tr><td
height=4 width=1 colspan=3><spacer type=block width=1
height=4></td></tr><tr><td width=6 rowspan=2><spacer type=block width=6
height=1></td><td valign=top><table border=0 cellspacing=0 cellpadding=0
width=100%><tr><td colspan=2><font face=verdana size=-2><b>Daryl
Johnston:</b></font></td></tr><tr><td><a href=s/55406><img
src=http://us.i1.yimg.com/us.yimg.com/i/my/sound.gif border=0 hspace=0 vspace=0
alt=audio width=16 height=13></a></td><td><font face=verdana size=-2><a href=s/
55406><b>Exclusive analysis</b></a></font></td></tr></table></td><td
valign=bottom align=right rowspan=2><a href=s/55407><img
```

```
src=http://us.i1.yimg.com/us.yimg.com/i/mntl/spo/03q1/mvpjackson.gif alt="MVP
Dexter Jackson" width=105 height=84 hspace=5 vspace=0
border=0></a></td></tr><tr><td><font face=verdana size=-2>&#183; <a href=s/
55408>Postgame News</a><br>&#183; <a href=s/55412>Box Score</a><br>&#183; <a
href=s/55407>Photos</a> - <a href=s/55411>Slideshow</a><br>&#183; <a href=s/
55409><b>More</b></a></font></td></tr></table></td></tr></table></td></tr></table></div>');
</script>
</center>
<small><small><br></small></small>
<table width=100% cellpadding=4 cellspacing=0 border=0 bgcolor=dfdfdf><tr><td
align=center><font face=verdana size=-2> <b>Y! Business Services</b> -
Visit the Yahoo! <a href=r/c9>Small Business Center</a></font></td></tr></table>
<table width=100% cellpadding=0 cellspacing=0 border=0 bgcolor=999999><tr><td
height=1><table cellpadding=0 cellspacing=0 border=0><tr><td
height=1></td></tr></table></td></tr></table>
<center><table width=95% border=0 cellspacing=0 cellpadding=2><tr>
<td width=30% valign=top nowrap><font face=arial size=-1>
<b>&#149;</b> <a href=r/h9>Web Hosting</a><br>
<b>&#149;</b> <a href=r/e9>E-commerce</a>
</font></td>
```

Figure 7. A very small portion of the HTML code for the Yahoo! home page in 2003 (`web.archive.org/web/20030127104212/www.yahoo.com/`). It constitutes an excellent example of all the evils that afflicted the web for some time: JavaScript generated content via `document.write`, intensive use of tables for layout, multitude of `font` elements, width and colours of tables specified in the markup… As a result, the amount of information contained in that piece of HTML is insignificant compared with the presentational code.

is read cell by cell, in order (for example, by a screen reader), it still makes sense. A more detailed and updated information about this concept is found on Techniques for WCAG 2.0 (Caldwell, Cooper, Reid & Vanderheiden, 2008, G57[1], F49[2]).

1 *Ordering the content in a meaningful sequence*, www.w3.org/TR/2008/NOTE-WCAG20-TECHS-20081211/G57

2 *Failure of Success Criterion 1.3.2 due to using an HTML layout table that does not make sense when linearized*, www.w3.org/TR/2008/NOTE-WCAG20-TECHS-20081211/F49

Anyway, the case is that using tables for layout does not necessarily mean that the so built website is not accessible, if they are used *judiciously* (and, certainly, a layout made up of nine nested tables is not what I would judge as sound).

Maintenance difficulty

It is difficult not to agree with this issue… until one sees some "pure" CSS layouts, which level of nested divs is even higher than that of the worst HTML table-based layouts. Nevertheless, as a general rule, it is true that deeply nested table-based layouts usually are a source of headaches when it comes to redesign, or even just to modify the content of a site, as is the case, for example, of the Hook Mitchell website that was shown in figure 3, as it is explained in Zeldman & Marcotte (2010, pp. 38–41), or the old version of Gilmore Keyboard Festival website reviewed in Zeldman (2003, pp. 49–53), although in these two cases it is more due to the use of images instead of text than for their table-heavy markup.

The major maintenance problem with tables with respect to CSS layouts is that whenever the layout changes, the markup of all the pages of the website must be changed too. This may not be an issue if a content management system (CMS) is being used in the backend and, anyway, as this thesis states, this is almost always the case also for CSS layouts, which often require changes in the markup in addition to the style sheet.

They break separation between presentation and content

This is the insurmountable problem of table-based layouts: since the visual structure of the page is hardcoded in the HTML document, it destroys any possible separation between presentation and content: the markup is closely bound to the final visual layout of the page. Note that this does not necessarily imply that CSS layouts are better (as it has been pointed out above and will be later demonstrated in this thesis, most current CSS layouts suffer the same problem), but it is more than enough to invalidate tables as a layout mechanism.

Why Are They Used for Layout?

Until now some of the frequently cited reasons against the use of tables for layout have been reviewed. But, if they have so bad *reputa-*

tion among web design community… why are they yet so used for layout purposes? Leaving aside browser inconsistencies in CSS support (something which this thesis is not concerned, since it assumes a best-case scenario of a perfect CSS 2.1 implementation), there is one reason that is is hardly debatable: as a general rule, *table-based layouts are much easier to achieve than their CSS equivalents*. As Budd has stated (2003):

> *I'm sure we've all found ourselves writing fairly complicated CSS to do something that would be trivial using tables.*

Furthermore, there are certain tasks that are not possible to do with CSS —at least, not without adding extra markup or altering the logical order of the content— that are very easy with tables; for instance, vertical alignment.

Cascading Style Sheets

In an attempt to redress the situation and return HTML to its origin as a language for structured documents, W3C developed Cascading Style Sheets (Lie and Bos 1996). Now, those presentational elements and attributes that had populated the web should have been removed from the HTML.

Cascading Style Sheets (CSS) are widely used nowadays, and they have contributed to recapture the original purpose of HTML, getting it rid of `FONT` tags and other presentational elements and attributes. But, as with any technology, using style sheets does not guarantee that we are separating presentation and content. One of the most common errors, specially among beginners, is the use of extra `div` elements and unnecessary classes and identifiers, which has been known as *classitis* and *divitis* (Zeldman, 2003, pp. 182 and 184).

Other problem of Cascading Style Sheets is that, as this thesis states —and will be demonstrated in further chapters—, in its current state it does not provide a true separation between presentation and content.

Conclusions

Other authors have already pointed out this problem. Thus, Meyer (2003a) says that, contrary to what some may have claimed or implied, it is not possible to create an arbitrary structure with no consideration toward its presentation. Others directly refers to this separation as a *myth* (Stain, 2000). This thesis assumes that hypothesis as a starting point, and chapters 5, 6, and 7 will demonstrate that, in effect, that separation is not currently possible, and that it is due to the lack of true layout mechanisms in CSS, as stated on *Chapter 8*. *Chapter 9* proposes the solution to solve that problem, adding to CSS a template-based layout mechanism, in an attempt to make Zeldman's words (2001) true:

> *We all know the future is about web standards. And web standards are about the separation of style from content —presentation from structure— design from data.*

3

What Layout Is

This chapter, as its title indicates, is aimed to provide the background on layout that is needed to understand what will come later. After defining which is the main function of graphic design, and some of the basic principles of layout, the chapter then goes on to the subject of grid systems, a classical concept of graphic design theory that, as will become apparent later, is essential to this thesis and to the solution proposed in it.

Introduction

Webster defines **layout** as "**1 :** the plan or design or arrangement of something laid out: as **a :** (dummy **5 b**) a set of pages (as for a newspaper or magazine) with the position of text and artwork indicated for the printer **b :** final arrangement of matter to be reproduced especially by printing"; and **lay out** as "**1 :** to put into a proper order or into a correct or suitable sequence, relationship, or adjustment".

For Wiktionary, layout is "*(publishing)* the process of arranging editorial content, advertising, graphics and other information to fit within certain constraints."

Similarly, Wikipedia, defines **page layout** as "the part of graphic design that deals in the arrangement and style treatment of elements (content) on a page".

Let us forget layout for a moment and concentrate on the discipline to which it belongs: *graphic design*. Next section offers a summarily introduction to it, enumerating and succinctly describing some of its basic elements and principles, before to explain what layout is and move on to the theory of grid systems. Needless to say, this is not —nor it *can* be— a treatise on graphic design, but, being this a dissertation that has the word "layout" on its title, it is compulsory to provide some background on this field.

Nevertheless, I have tried to be as concise as possible and describe only those aspects of graphic design that have some bearing on the matter of study of this thesis, and on the proposed solution.

What Is Graphic Design?

> *To understand the meaning of design is… to understand the part form and content play… and to realize that design is also commentary, opinion, a point of view, and social responsibility. To design is much more than simply to assemble, to order, or even to edit; it is to add value and meaning, to illuminate, to simplify, to clarify, to modify, to dignify, to dramatize, to persuade, and perhaps even to amuse.*
>
> —Paul Rand (as cited in Samara, 2007, p. 6)

> *Design is not just what it looks like and feels like. Design is how it works.*
> —Steve Jobs (2003)

If we asked professional designers what is the main function of graphic design, no doubt we would obtain many different answers, but, surely, most of them would be variations of the same essential idea: *to communicate*. Thus, graphic design is not (*only*) "to make things look *pretty*" (despite the many comments in response to McWade's question (2009) that express in such terms), but *to convey a message*, using tools such as images and typography: "graphic design is a creative process that combines art and technology to communicate ideas" (Poggenpohl, 1993); "a graphic designer is a communicator: someone who takes ideas and gives them visual form so that others can understand them" (Samara, 2007, p. 6); "a truly effective graphic communication is the combined result of both message content and message presentation" (Brahmachari, 2000). In other words, the purpose of graphic design (and what differentiates it from other disciplines in visual arts) is not to alter the *message* (which is defined by the client, does not emanate from the designer), but to *clarify* it and communicate it effectively.

But, what are the tools with which the graphic designer counts to carry out his purpose? The paragraph that follows to the quote above by Samara (Samara, 2007, p. 6) give us some hints about it:

> *The designer uses imagery, symbols, type, color, and material —whether it's concrete, like printing on a page, or somewhat intangible, like pixels on a computer screen or light in a video— to represent the ideas that must be conveyed and to organize them into a unified message.*

Samara, in the preceding quote, has enumerated some of the basic building blocks of graphic design. Graphic design theory distinguishes between **elements** and **principles** of graphic design. Although the concrete elements and principles may vary depending on the sources, most books and syllabi of introductory courses agree on at least the following basic elements of design: *line*, *shape*, *form*, *space*, and *colour*. Images and type are not formal elements of design, but they both can function as such in page layout (Evans, 2005, pp. 40 and 42).

As it has been stated in the introduction of this chapter, this dissertation is not the right place to describe them, but there is one that, because of its importance, deserves some discussion: *typography* (or *type*). Next section explains its close relationship with layout and mentions some differences between print and web typography. As for the principles, since they are closely related with layout, they will be described in that section.

Typography

> *Typography exists to honor content.*
> —Bringhurst (2005)

Typography is sometimes misconceived as the font types used in a text. But, notwithstanding that is, of course, an essential component of any design, typography is much more than *simply* choosing the right typefaces for a project. It fact, in graphic design theory layout is usually considered a subset of typography. Thus, Cullen (2005, p. 101), differentiates between *macro* and *micro* perspectives in typography: whereas by taking a micro perspective the designer concentrates on details such as font type and size, kerning, spacing, hyphenation, punctuation marks, etcetera, the macro view deals with the overall design layout.

I am not going to enter into details about these micro typographic issues because, despite the undoubtedly importance that they have, both in printed and electronic media, they usually do not present difficulties to be well defined in CSS, with some exceptions, from which is specially notorious the absence of support in CSS2 for *hyphenation*, which, together with the lack of control over screen resolution, the ability of web users to resize text, and the usually poor paragraph-breaking algorithms of web browsers, makes justification on the web almost impracticable. Vertical alignment of text is decidedly another issue in CSS, too.

Another important difference between print and web typography is the limited fonts that a designer can *safely* use on the web, when compared to the plethora of fonts available in traditional print design (Lie, 2007). Although CSS has provided support for web

fonts through the `@font-face` construct introduced in CSS2 (Bos, Lie, Lilley & Jacobs, 1998), due to legal issues, it has not been until very recently when they are becoming a reality thanks to initiatives like Typekit[1] or FontSkirrel[2].

Line Length

Line length has an extraordinary effect on the *legibility* of the page (which is, after all, the ultimate goal of typography). According to Bringhurst (2005, p. 26), "the 66-character line (counting both letters and spaces) is widely regarded as ideal", although anything from 45 to 75 characters is usually considered satisfactory. It could be argued, though, that Bringhurst's work, as authoritative as it is, only refers to printed, *paged* media, and, therefore, its conclusions can not be directly translated as such to the web.

Mills & Weldon (1987), in their review of empirical studies concerning the readability of text from computer screens, did not reach any definitive conclusion, apart than the fact that more research on this field was needed. It must be also considered that the features of existing CRT displays by the time their review was conducted had nothing to do with modern monitors, neither in quality nor size. However, one of the conclusions that they draw as working hypotheses, is that "text with 80 characters per line on a screen width seems easier to read than text with 40 characters per line on a line of the same length" (p. 353). More recent studies seem to confirm that reading on the screen calls for slightly longer lines than on paper. Thus, Shaikh (2005), in a study about the effects of line length on reading online news, found that reading rates were fastest at 95 characters per line than at 35, 55, or 75 (the other line lengths considered). However, his study must be taken cautiously, since it was made with only twenty participants. Bernard, Fernandez & Hull (2002), in a similar study, did not found significant differences in reading time or reading efficiency between full, medium, and short line lengths, although adult participants preferred medium and narrow length lines, whereas most children opted for narrow lines. As

1 typekit.com
2 http://www.fontsquirrel.com/

Wilkinson (2009) and Weinschenk (2003) has suggested, there is not yet enough research as to establish rigid guidelines with regards to line length for web pages. In addition, as had already been stated in the review of Mills & Weldon (1987, p. 341), there are many other formatting factors that influence readability, like typeface and font size, line height, or colour. Anyway, whatever be the *ideal* line length on the web, it seems obvious that any layout mechanism should allow the author to specify the width of the layout in terms of the font size.

Layout

Although some general definitions of layout have been provided in the opening section of this chapter, its crucial importance to this thesis makes it deserve a more detailed explanation. First, a more formal definition comes from Ambrose & Harris (2005):

> *Layout is the **arrangement** of the **elements** of a **design** in relation to the space that they occupy and in accordance with an overall aesthetic scheme. This could also be called the management of **form** and **space**. The primary objective of layout is to present those **visual** and **textual** elements that are to be **communicated** in a manner that enables the reader to the receive them with the minimum of effort. With good layout a reader can be navigates through quite complex information, in both print and electronic media.*

The mission of layout is therefore to create a clear visual hierarchy that matches the logical structure of the message to be communicated, be it a book, a magazine cover, a poster, a business card, or anything else. To accomplish this task, the designer relies on some design principles, or compositional factors, which, combined with type, colour, texture, and the rest of basic elements of design, contribute to create a visual structure that conveys the message and is appealing for the reader. Although there is no consensus on which these principles of design are, some of them are the most frequently cited as the most relevant for layout, and will be briefly reviewed in the following subsections. But, before to describe them, it is worth

to mention some fundamental rules upon which the theory of layout in graphic design is built.

Gestalt Principles

Most layout principles are based on a set of laws and principles collectively known as the *Gestalt principles of perception*. Gestalt theory was developed by Austrian and German psychologists in the late 1800's and the early 1900's, and provide a theoretical background that explains human perception and our tendency to *group* things (Graham, 2008). Gestalt visual laws provide scientific validation of compositional structure, and are therefore an essential part of any graphic design curriculum. Lynch and Horton (2009) cite the following gestalt principles among the most important for page layout:

Proximity

Elements that are close to each other are perceived as more related than elements that lie farther apart (see figure 1.1).

Similarity

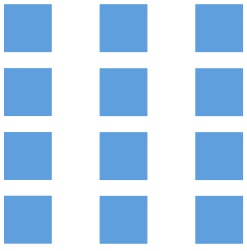
Viewers will associate and treat as a group elements that share consistent visual characteristics. In figure 1.2 it can be appreciated how similarity (or, in this case, *dissimilarity*) can be achieved through the use of several elements of design, such as *shape* (first row is perceived as different from the others because it is made of circles instead of squares) or *colour* (third row).

Continuity

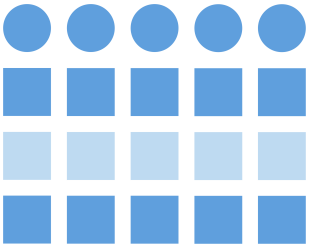
Humans prefer continuous, unbroken contours and paths, and the vast majority of viewers will interpret figure 1.3 as two crossed lines, instead of four lines meeting at a common point.

Closure

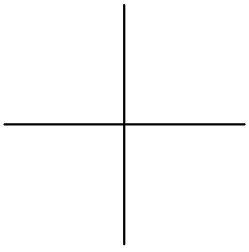
We have a powerful bias to see completed figures, even when the contours of the figure are broken or ambiguous. For example, in figure 1.4 we see a white rectangle overlying four circles, and not four circles, each having a section missing.



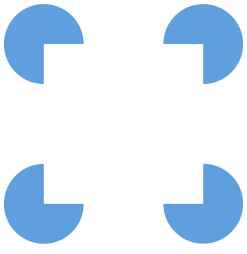
#1 **Proximity** between elements causes us to see columns.



#2 **Similarity** causes us to see related horizontal rows.

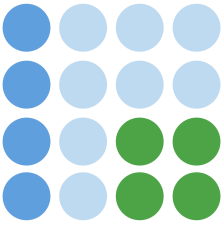


#3 **Continuity** is the reason why we prefer to see two lines crossing, instead of four lines meeting in the middle.

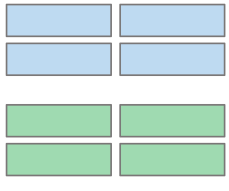


#4 **Closure** causes us to see a white square, instead of four broken circles.

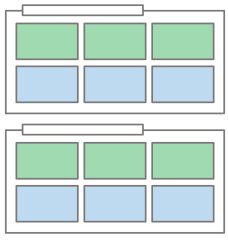
#5 The classic Rubin Vase illusion is an example of **figure-ground relationships**: either a vase or two face profiles can be seen in the figure. As the relative size of the vase increases, it tends to dominate over the faces.



#6 **Uniformity** allows us to see a blue column and a green group.



#7 **Uniformity** is a common mechanism for organising user interfaces.



#8 **Uniformity**, **enclosure**, and **proximity** help to distinguish groups.

Figure 1. Some of the most relevant Gestalt principles for page layout.

Figure-ground relationships

In figure-ground reversal the viewer's perception alternates between two possible interpretations of the same visual field: either a goblet or two faces can be seen in figure 1.5, but both can not be seen at once. **Proximity** has also a strong effect on figure-ground relationships: it is easier for most people to see the goblet when it is wider and the faces are farther apart. Also, visual elements that are relatively small will be seen as discrete elements against a larger field. The small element will be seen as the *figure* and the larger field as the *ground* around the figure.

Uniformity

Uniformity refers to relations of elements that are defined by enclosing elements within other elements, regions, or discrete areas of the page (see figures 1.6, 1.7, and 1.8).

How We Read a Page

As it has been stated in introduction of this section, one of the crucial aspects of layout is that, with it, the designer establishes a hierarchy among the elements of the page, giving more prominence to the most important elements, and guiding the eyes of the reader through the overall design. But, although the designer can thus influence the visual structure and the order in which the different elements of the design are perceived, even in the absence of a conscious layout effort by the designer (let us think, for example, in a dull page of text), there are certain areas of a page that are some areas of the page that are more *active*, whereas others are more *passive*.

Specifically, humans (in western culture) tend to start scanning a page at the top left corner and then follow a diagonal path until the right bottom corner. This well-known pattern is shown in figure 2, due to Ambrose & Harris (Ambrose & Harris, 2008, p. 14).

Interestingly, the above mentioned pattern does not apply exactly equal to the web. This is probably because, as has been repeatedly said by usability experts, *people do not read web pages*, but *scan* them (Nielsen, 1997a, 2000, p. 104; Krug, 2000, p. 22). *Eyetracking*

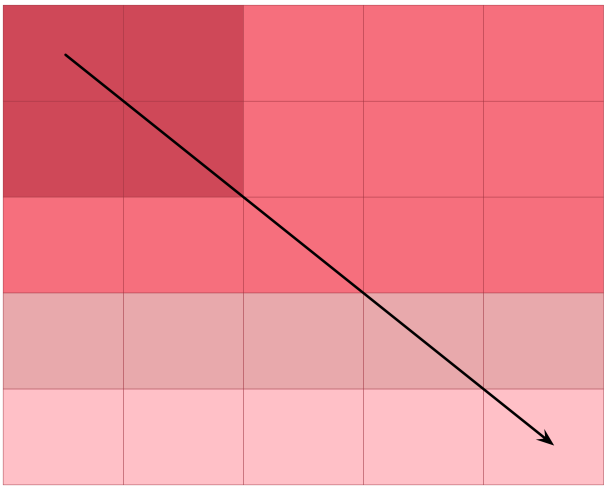


Figure 2. When faced with a new page of information, the human eye habitually looks for an entrance at the top left and scans down and across to the bottom right corner (Ambrose & Harris, 2008, p. 14).

Figure 3. Users tend to scan web pages following an F-pattern. Some authors also refer to a "golden triangle', highly dependent on the fold point of the page.

studies (Nielsen, 2006b) have proved that users on the web scan pages following what has been named an **F-pattern**, first quickly scanning across the top from left to right in two stripes, and then scanning down the page as they rapidly move forward in search of something meaningful (Ambrose & Harris, 2008, p. 18). When combined with page *fold* (the imaginary line that limits what the user can see before having to scroll down), gives as a result the pattern depicted by Lynch and Horton (2009) in figure 3.

But the F-pattern can not be considered an absolute truth. First, as Lynch and Horton (2009) have pointed out, Nielsen's study (2006b) is biased toward web pages dominated by text information, and, as users are learning to identify standard components of web pages, such as navigation, shopping cart, search, etcetera, new trends are emerging that will eventually change the *best practices* for page layout on the web.

In addition, as it happens in print media, web designers have the capability of changing these common patterns employing the same principles of layout that are used in traditional graphic design.

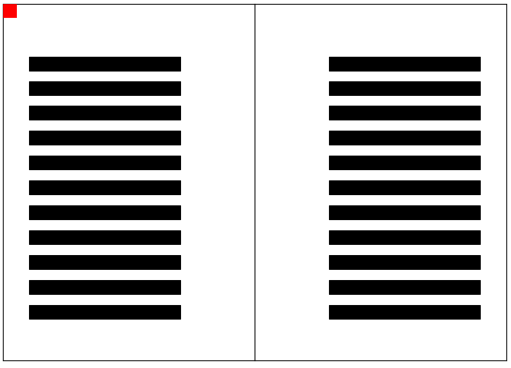
Hierarchy

Whereas design elements, such as colour, form, image, space, and typography collaborate to convey the intended message, it is when the designer activates the page through the placement of the visual elements and all the components are clearly interrelated that the whole design makes sense. That ordering system, or *hierarchy*, defines the importance for every visual element and determines their sequence through the design (Cullen, 2005, p. 73).

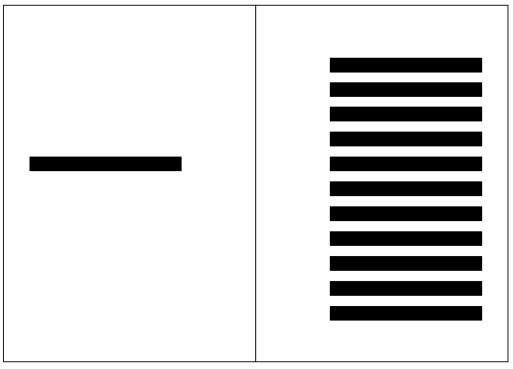
Hierarchy can be achieved with an appropriate use of design elements and other design principles, such as *alignment*, *scale*, or *emphasis*.

Grid Systems

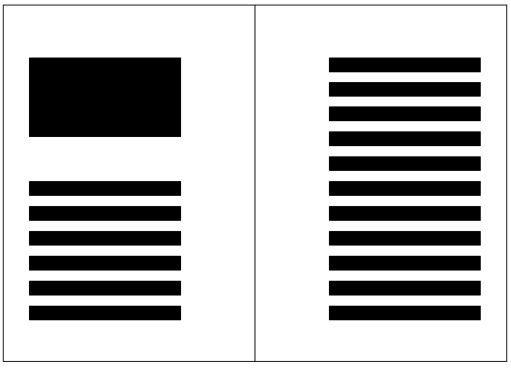
Simply stated, a grid is no more than a series of intersecting axes that create horizontal and vertical divisions of space on the page (Cullen, 2005). Although neither the grid nor even graphic design were



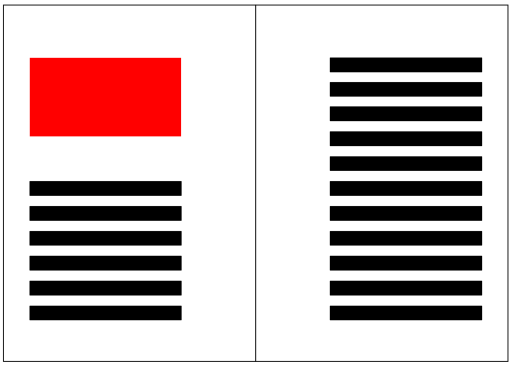
Neutral This illustration shows a neutral page with no hierarchy between the two text columns. A reader will naturally enter the design at the top left.



Position An obvious placement of a design element introduces a hierarchy, such as this lone heading on the verso page.



Position and size Positioning an element in the entry hotspot while altering its size and introducing spacing establishes its dominance in the hierarchy.



Position, size, and emphasis Another technique is to add extra emphasis to an element to cement its position at the top of the hierarchy, as seen in the use of colour above.

named as such until the mid-twentieth century, its use dates from antiquity, and can be found in ancient art, architecture, urbanism, and other fields. One of the most notorious modern uses of grid in urban city planning is that of Barcelona Eixample, shown in figure 5.

Grids were born in Switzerland after World War II, when the first examples of printed matter designed with the aid of a grid appeared, as a response to the chaotic layouts that Industrial Revolution had brought: images, advertisements, photographs, along with

Figure 5. The Eixample of Barcelona is one of the most evident uses of grids in city planning. It was conceived by the visionary Ildefons Cerdà in the 1850's as an expansion of Barcelona between the old city and the surrounding small towns. It consists of straight streets crossed by a long wide diagonal, and building blocks of even size (113 meters each side), with its characteristic chamfered street corners. Picture from Institut Cartogràfic de Catalunya (`www.icc.es`).

an increasing array of typefaces, suddenly competed for attention, with no precedence in the classical book (Roberts, 2007, p. 13). This was identified as a problem that had to be solved. But the *grid* term did not yet exist as such. It was first named and described by Müller-Brockmann (1971), although it was not until the seminal book of the same author (1981) when the grid construction was systematically explained.

Elements of a Grid

Every grid, no matter how complex it may become, is made of the same basic elements, which are: *margins*, *markers*, *columns*, *flowlines*, *spatial zones*, and *modules* (see figure 6). These elements can be combined or omitted from the overall structure to accommodate to the needs of each concrete design (2009):

- **Margins** represent the amount of space between the trim size, including gutter, and the page content. Margins can also house secondary information, such as notes and captions.
- **Markers** help a reader navigate a document. Indicating placement for material that appears in the same location, markers include page numbers, running heads and feet (headers and footers), and icons.

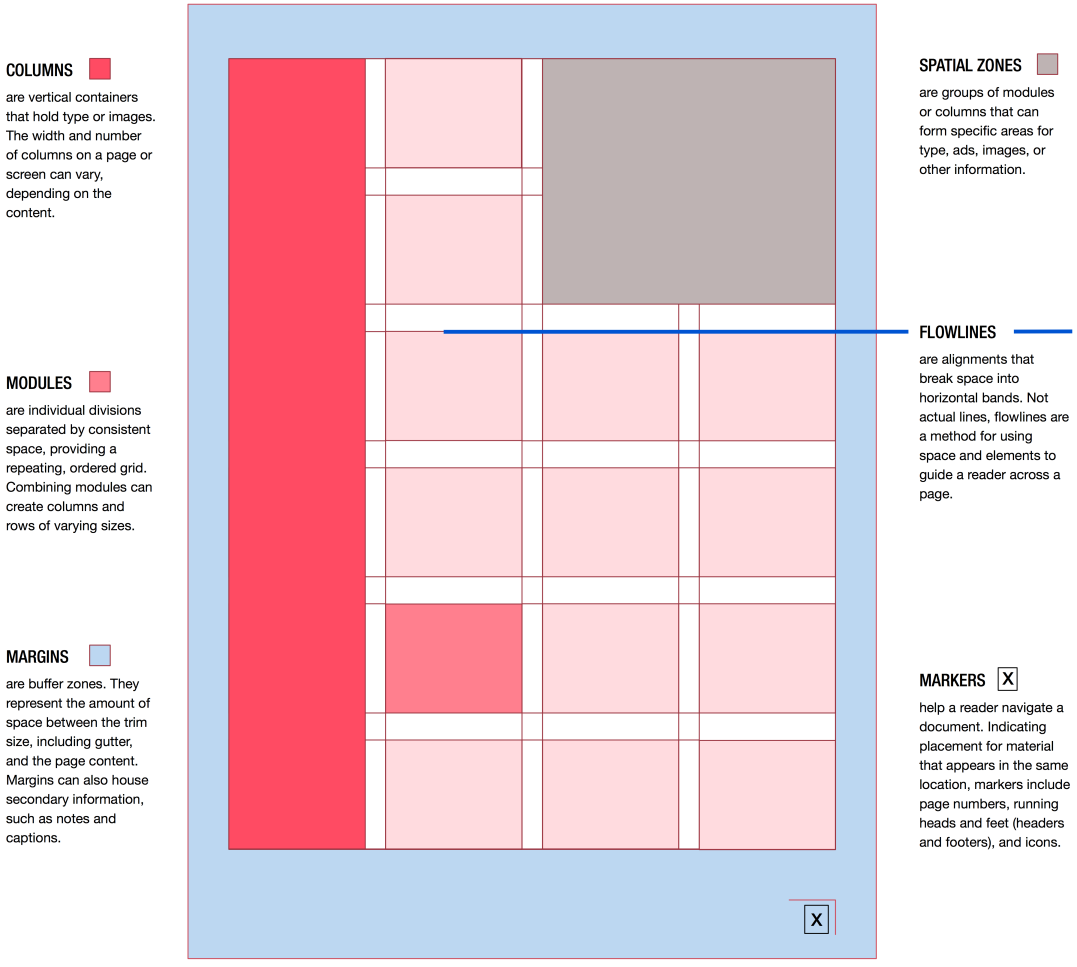


Figure 6. The main components of a grid (margins, markers, columns, flowlines, spatial zones, and modules) are depicted in context (within a grid) and briefly explained. Figure and descriptions are from Tondreau (2009).

- **Columns** are vertical containers that hold type or images. The width and number of columns on a page or screen can vary, depending on the content.
- **Flowlines** are alignments that break space into horizontal bands. Not actual lines, flowlines are a method for using space and elements to guide a reader across a page.

- **Spatial zones** (also known as *modules*) are alignments that break space into horizontal bands. Not actual lines, flowlines are a method for using space and elements to guide a reader across a page.

Types of grids

There are as many types of grids as designers. However, it is possible to identify and classify a few basic grid structures.

- A **single-column grid** is generally used for continuous running text, such as essays, reports, or books. The main feature on the page or spread is the block of text.
- A **two-column grid** can be used to control a lot of text or to present different kinds of information in separate columns. A double-column grid can be arranged with columns of equal or unequal width. In ideal proportions, when one column is wider than the other, the wider column is double the width of the narrow column.
- **Multicolumn grids** afford greater flexibility than single or two-column grids, combine multiple columns of varying widths and are useful for magazines and websites.
- **Modular grids** are best for controlling the kind of complex information found in newspapers, calendars, charts, and tables. They combine vertical and horizontal columns, which arrange the structure into smaller chunks of space.
- **Hierarchical grids** break the page into zones. Many hierarchical grids are composed of horizontal columns.

4

Layout Languages

Whereas this thesis is focused on Cascading Style Sheets, and so is the proposed solution, the layout issues addressed on it are by no means new. Not only from the graphic design perspective, for which the previous chapter have provided the background on layout and grid systems on which the proposed solution is based, but the subjects of layout in general, and document formatting in particular, have had a long tradition in Computer Science, from graphical user interfaces to document engineering or electronic publishing.

Introduction

Although, apparently, the languages reviewed in this chapter may seem somehow disconnected, or even arbitrarily chosen, their purpose is to provide a review of how the issue of layout is addressed in them. Specifically, the languages reviewed in this chapter belong to the following categories:

- User interface languages
- Graphical libraries of programming languages

User Interface Languages

By *user interface languages* I mean whatever language specifically oriented towards the creation of graphical user interfaces.

XAML

XAML (*Extensible Application Markup Language*) is a markup language for *declarative application programming* (Microsoft, 2009b). It has been created by Microsoft and is built into the Windows Presentation Foundation (WPF), which, in turn, is the platform for building the graphical user interface of both client and browser-based applications, inside .NET framework.

WPF has an extensive set of features that include vectorial graphics, 3D rendering, audio and video players, or even animations, among many others. It is worth to mention the distinction that WPF makes between *flow documents* and *fixed documents*, resembling what some desktop publishing tools do.

But what is relevant to this thesis are the layout capabilities that XAML has to offer. Although this dissertation is not the place for explaining in detail how it works, its main controls will be described, so that they can be later compared with the layout features of CSS, as well as with the solution proposed by this thesis.

Layout System Overview

The layout system in Windows Presentation Foundation is based on *relative positioning*. This must not be confused with the relative positioning in CSS. Whereas in the latter, the term *relative* means

that a so positioned box is *shifted* with respect to its original position in the normal *flow* of the document (Bos et al., 2009, §9.4.3), relative positioning in WPF refers to the ability of the layout system *to adapt to changes in window size and display settings* (which in CSS are called *liquid* and *elastic* layouts, described on *Chapter 6*, on pages 112 and 113, respectively). More specifically, in Windows Presentation Foundation, the layout of elements on the screen is the result of a negotiation between the control to be rendered and its parent (Microsoft, 2009a, *Layout* section):

1 First, the control tells its parent what location and size it requires.
2 Secondly, he parent tells the control what space it can have.

As the documentation itself acknowledges, this may be an intensive process, being greater the number of calculations made the larger the number of children elements is (Microsoft, 2009c, *The Layout System* section), because, even in its simplest form, layout is a recursive process in which, for each child of the control to be rendered, at least two passes are performed: one for measuring the size of the element, and another for arranging it in its container.

Layout Controls

Windows Presentation Foundation (and therefore XAML) provides the following controls for most common layouts:

`Canvas`

Child controls provide their own layout.

`DockPanel`

Child controls are aligned to the edges of the panel.

`Grid`

Child controls are positioned by rows and columns.

`StackPanel`

Child controls are stacked either vertically or horizontally.

`VirtualizingStackPanel`

Child controls are virtualized and arranged on a single line that is either horizontally or vertically oriented.

`WrapPanel`

Child controls are positioned in left-to-right order and wrapped to the next line when there are more controls on the current line than space allows.

Some of them are the typical controls that we can find on graphical libraries of many programming languages, like the canvas, which defines an area within child elements are positioned using coordinates, very similar to what can be achieved in CSS using *absolute positioning*. And, like absolute positioning too, it is the simplest control to implement, since it does not have associated any layout policy: the user (the programmer, in this case) is responsible for controlling the position and dimensions of child elements, and they are never resized. `VirtualizingStackPanel`, and its counterpart, `VirtualizingStackPanel` (the *virtualizing* prefix merely refers to an optimisation feature of WPF under which only those elements that are visible on the screen are generated), are also simple controls that allows child elements to be positioned one after another, either horizontally (in a *row*) or vertically (in a *column*), which might be compared (to a certain extent) to the way in which *inline* and *block* boxes, respectively, are laid out in CSS (although these controls provide some additional features that are not possible in HTML/CSS, like *logical scrolling*, namely, the ability to scroll to the next element, instead of the physical scrolling to which we are used to on the web). This is also the case of `WrapPanel`, which behaves very similar to the normal flow of CSS, positioning its children in sequential position from left to right, breaking content to the next line at the edge of the containing box.

`DockPanel`, on the other hand, allows to position elements relative to both other elements within the same container and the edges of the container (by means of the property `Dock` and the values `Left`, `Top`, `Right`, and `Bottom`). In addition, it permits the last element to

fill the remaining space. Although, with the possible exception of this last feature, the layout provided by this control is not nothing that can not be achieved in CSS with a combination of floats, positioning, margins, and the rest of mechanisms that will be reviewed on *Chapter 5*, it is worth to include an example to later compare how the same layout is much easier to achieve using this control in XAML than with HTML & CSS. Such a example is provided in figure 1, which shows the XAML code that gives as a result the layout shown in figure 2.

`Grid` control on WPF provides a mechanism for creating *grids*, the graphic design system that was introduced on *Chapter 2*.

The most interesting control for the purposes of this thesis is also the most flexible (and, for that very reason, the most complex) of all the layout components of Windows Presentation Foundation: the `Grid`. While the layouts allowed by the aforementioned controls can be reproduced, to a greater or lesser extent, in CSS, this control provides a convenient way to deal with *grids*, something that, despite the interest that this subject has caused in the web standards community in recent years, is not possible with CSS.

Grids on XAML

Although I am not going to dwell on details about the multiple features of `Grid` control and the possibilities it brings to create complex layouts, there are, though, several aspects that must be at least mentioned, because they closely match the solution purposed in this thesis to the problem of layout on the web.

But, what does this control essentially do? Basically, it allows the designer to define a structure of *rows* and *columns*. Although, at first glance, it could be thought as an HTML table, there is a substantial difference: the content is not placed inside the cells defined by the grid. Instead, the grid only defines how the layout is (how many rows and cells it has). It is later, once the grid have been defined, when the content is assigned a position corresponding to a cell in the grid (cells can span several rows and columns, and this is indicated for each concrete block of content too). This can be easily observed in the XAML code of figure 3, which defines a grid of four rows and three columns, where the first and last rows span the three columns (the resultant layout can be seen in figure 4).

Figure 1. An XAML page that uses a `DockPanel` control to create the layout shown in figure 2.

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
WindowTitle="DockPanel Sample">
  <DockPanel LastChildFill="True">
    <Border Height="25" Background="SkyBlue" BorderBrush="Black"
BorderThickness="1" DockPanel.Dock="Top">
      <TextBlock Foreground="Black">Dock = "Top"</TextBlock>
    </Border>
    <Border Height="25" Background="SkyBlue" BorderBrush="Black"
BorderThickness="1" DockPanel.Dock="Top">
      <TextBlock Foreground="Black">Dock = "Top"</TextBlock>
    </Border>
    <Border Height="25" Background="LemonChiffon" BorderBrush="Black"
BorderThickness="1" DockPanel.Dock="Bottom">
      <TextBlock Foreground="Black">Dock = "Bottom"</TextBlock>
    </Border>
    <Border Width="200" Background="PaleGreen" BorderBrush="Black"
BorderThickness="1" DockPanel.Dock="Left">
      <TextBlock Foreground="Black">Dock = "Left"</TextBlock>
    </Border>
    <Border Background="White" BorderBrush="Black" BorderThickness="1">
      <TextBlock Foreground="Black">This content will "Fill" the remaining
space</TextBlock>
    </Border>
  </DockPanel>
</Page>
```

Grid Units

Three unit types are allowed for specifying the width and height of columns and rows, respectively:

`Auto`

The size is determined by the size properties of the content object (it is the default behaviour that has been described above).

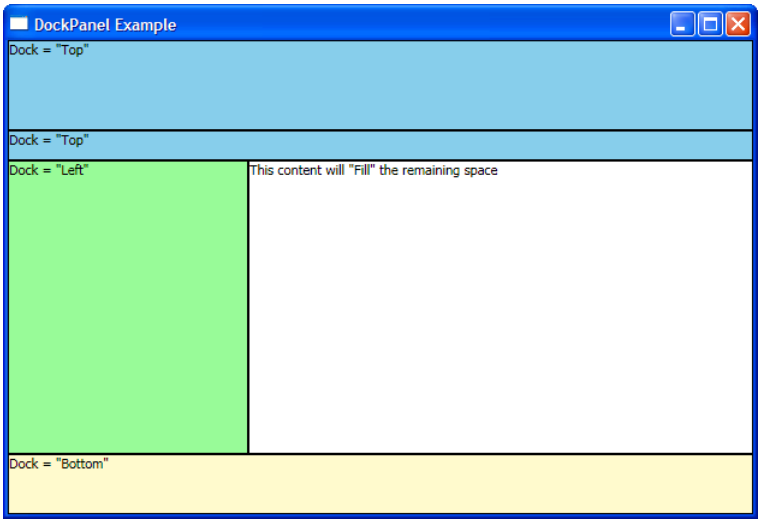


Figure 2. A WPF window that uses a `DockPanel` control to lay out its components. Child elements are arranged based on the order in which they are defined in the XAML source code. Therefore, they are relative to other elements, but also to the edges of the container, by means of the `Dock` property of this class, which allows to align a component to the left, top, right, and bottom of the parent, in a somewhat similar manner to what `float` property does in CSS.

Pixels

If a number is specified as the width or height value of a column or row of the grid, it represents a size in pixels.

** (star)*

The value is expressed as a weighted proportion of available space.

The last value, *star* (or *asterisk*), is the most interesting for the purposes of this thesis, because it allows to specify different levels of *flexibility* for row heights and column widths. For instance, in the example above, by setting a height of * to the third row, cells on that row adapt to the height of the window, enlarging or shrinking as needed to fill all the available height. There is nothing similar to this ability currently in CSS.

XUL

XUL (XML User Interface Language) is a markup language created by Mozilla for building the user interface of cross-platform applications. It does not contain so many features as XAML, although the purpose of both languages is similar: to provide a manner of defining the graphical interface library of an application in a declarat-

```
<Window x:Class="GridLayout.GridWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Grid Layout Example in XAML" Width="768" Height="480">
  <Window.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="FontSize" Value="24"/>
    </Style>
  </Window.Resources>
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="*"/>
      <RowDefinition Height="100px"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <TextBlock Grid.ColumnSpan="3" Background="#CBF50F">This text spans the
whole first row</TextBlock>
    <TextBlock Grid.ColumnSpan="2" Grid.Row="1" Background="#C5E88A"
TextWrapping="Wrap">Lorem ipsum ... </TextBlock>
    <TextBlock Grid.RowSpan="2" Grid.Row="1" Grid.Column="2"
Background="#FF333838" Foreground="White">This is the sidebar</TextBlock>
    <TextBlock Grid.Row="2" Grid.Column="0" Background="#799666"
TextWrapping="Wrap">A single cell</TextBlock>
    <TextBlock Grid.Row="2" Grid.Column="1" Background="#9CD989">Another
one</TextBlock>
    <TextBlock Grid.ColumnSpan="3" Grid.Row="3" Background="#95AB3C">This is
the footer</TextBlock>
```

```
    </Grid>
  </Window>
```

Figure 3. XAML code for the grid layout shown on figure 4. Note that first, the grid is defined, inside the element `Grid`, and then each piece of content indicates its position into the grid by means of `Grid.Row` and `Grid.Column` attributes. Interestingly, is also each content element that defines how many rows and columns it spans, instead of leaving that decision to the grid itself, as is done in the Template Layout Module, the solution proposed in this thesis for CSS3. This resembles the notion of *modules* in grid system theory (described on *Chapter 3*, p. 53).

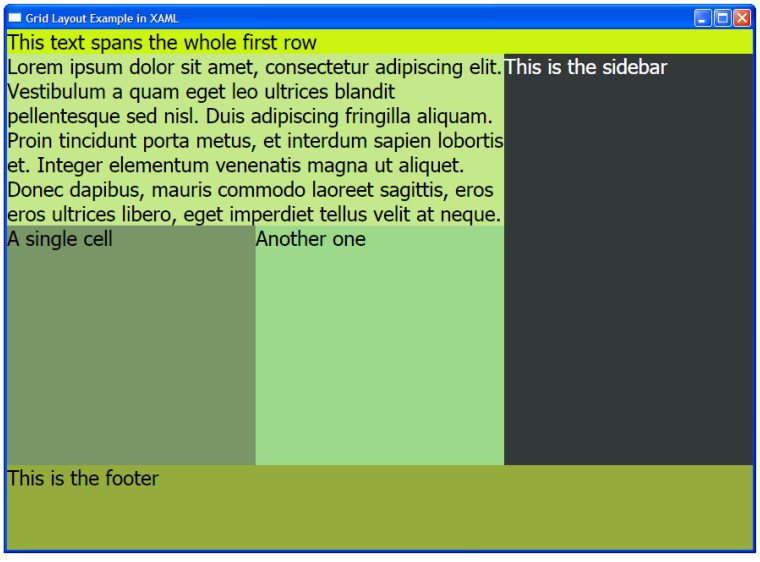


Figure 4. A WPF window that uses a `Grid` control to create a grid of four rows and three columns (4×3) and then lay out some content into the cells of the grid.

ive way, so that they are independent of the programming language (and, in the case of XUL, of any platform, too, since it allows to create the GUI of an application for any operating system in which it runs, without depending on any specific native window system).

To accomplish this task, XUL offers controls for the most common widgets of graphical user interfaces: windows, buttons, input fields, toolbars, lists and tables (*trees*, in XUL terminology), etcetera. In addition, it provides support for scripting, and integration with programming languages through XPCOM and XPConnect.

Again, these are not the pertinent features for this research. As with XAML, what this thesis is concerned is about the layout capab-

ilities of XUL, in order to be compared to those of CSS. Therefore, the following pages are aimed to briefly review such layout features.

The Box Model

The simplest layout components in XUL are boxes, which allow to lay out elements horizontally or vertically.

As in CSS, layout in XUL is based on boxes inside boxes. In its most basic form, a box allows lay out its children in one of two orientations, either horizontally or vertically. Various attributes placed on the child elements in addition to some CSS style properties control the exact position and size of the children (Mozilla, 2007).

```
<hbox>
  <!-- horizontal elements -->
</hbox>

<vbox>
  <!-- vertical elements -->
</vbox>
```

In addition to `hbox` and `vbox`, XUL also contains a generic box that can work like an horizontal or a vertical box just changing the value of its `orient` attribute, which may be useful if the orientation of the box needs to be changed dynamically, from JavaScript or any other programming language.

It is possible to achieve many different layouts with only these two types of boxes, simply by nesting them according to our needs. The extra flexibility comes from several attributes of these elements, of which the most important for layout are the following:

`align`

The `align` attribute specifies how child elements of the box are aligned when the size of the box is larger than the total size of the children. Note that for boxes that have horizontal orientation, it specifies how their children will be aligned *vertically*, and vice versa: for vertical oriented boxes, it specifies how their children are aligned *vertically*. Its possible values are: `start`, `center`, `end`, `baseline` (horizontal boxes only), and `stretch`.

While the meaning of the other values is quite obvious, it is worth to briefly explain how the **`stretch`** value affects the layout of the child elements of the box to which it is applied, *causing them to*

stretch until fit the size of the box. Using it, it is very easy to achieve, for instance, equal-height columns, something that is only possible in CSS using workarounds.

`pack`

It is the opposite of the `align` attribute: it allows to align the child elements of the box when it is larger than the size of the children, but, in this case, it defines the horizontal alignment when it is applied to boxes with horizontal orientation, and the vertical alignment for vertically oriented boxes. Only admits three values: `start`, `center`, and `end`.

`flex`

This attribute is the most interesting for the purposes of this review, since, although for element widths it behaves like percentages in CSS, there is nothing similar in CSS for the height of the elements. It defines the *flexibility* of an element, which, according to the XUL reference (Mozilla, 2009d), "indicates how an element's container distributes remaining empty space among its children. Flexible elements grow and shrink to fit their given space. Elements with larger flex values will be made larger than elements with lower flex values, at the ratio determined by the two elements."[1]

In other words, this attribute behaves like the *star* (`*`) did for width and height values in XAML: using `flex` it is possible to define constraints for row heights and column widths, so that they are *flexible* (that is, they adapt to the size of their container) but retaining the proportions among them. Thus, let it be the following code:

```
<vbox flex="1">
   ...
</vbox>
<vbox flex="2">
   ...
</vbox>
```

1 Mozilla (2008). flex. Retrieved from https://developer.mozilla.org/en/XUL/Attribute/flex/

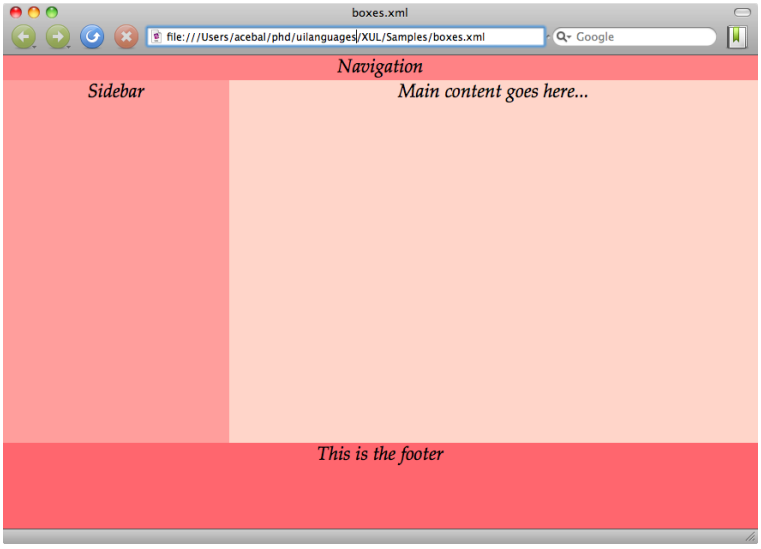


Figure 5. Combining boxes with different orientations and flexibilities is possible to recreate in XUL a similar layout to that shown in figure 4.

```
<vbox height="185">
   ...
</vbox>
<vbox height="1">
   ...
</vbox>
```

It defines four rows, of which the first, second, and fourth are flexible, the second being the double of the first and fourth, which are of equal height. Note that the third row has an explicit height of 185 pixels, and that is what makes this property so interesting for the purposes of this thesis: it allows to mix fixed and flexible rows and columns, and the flexible ones automatically adapt to the remaining room, once discounted the size of the fixed ones, something that is *not currently possible in CSS*.

Once the main properties of boxes have been enumerated and described, they will be put together in the following example, which reproduce a layout very similar to that that has been done earlier in this chapter with XAML.

```
<?xml version="1.0"?>
<?xml-stylesheet href="css/boxes.css" type="text/css"?>
<!DOCTYPE window>
<window xmlns="http://www.mozilla.org/keymaster/
gatekeeper/there.is.only.xul"
        xmlns:html="http://www.w3.org/1999/xhtml"
title="Boxes in XUL">
  <vbox flex="1">
    <description id="header">This is the
header</description>
    <description id="nav">Navigation</description>
    <vbox flex="1">
      <hbox flex="1">
        <description id="sidebar"
flex="1">Sidebar</description>
        <description id="main-content" flex="2">Main
content goes here...</description>
      </hbox>
      <description id="footer" height="100">This is
the footer</description>
    </vbox>
  </vbox>
</window>
```

To recreate exactly the XAML layout of figure 4 in XUL would suffice to divide the main content in two horizontal boxes; the second one, in turn, in two vertical boxes; and, finally, to alter their position. This is a very intuitive process, when compared with CSS, as it will be shown in the two following chapters.

Graphical Libraries of Programming Languages

Java Swing

Swing is the library for building graphical user interfaces (GUI) in Java. It is part of the Java Foundation Classes (JFC), which is the

graphical framework that implements all the GUI functionality in Java, and also comprises AWT and Java 2D.

A Brief Historical Note

Java was conceived as a multi-platform and multi-device programming language. In 1991, the members of an small project at Sun that would eventually lead to the Java programming language that was announced to the world in 1995, had come to the conclusion that "at least one significant trend would be the convergence of digitally controlled consumer devices and computers" (Byous, 2003), and the predecessor of Java was in fact shown in 1992 in a handheld home-entertainment device.

This short piece of history is relevant because the requirements of the Java programming language were similar to those of the web itself: multiple devices, operating system independent… Therefore, among many other considerations, it was needed a manner to build graphical user interfaces that did not depend on the underlying window environment. Not controlling the specific features of the device where the program would run also meant that the programmer should be able to create layouts that adapted to the dimensions of the screen. Although the initial GUI library of Java was AWT, not Swing, the aforementioned requirements are the same, and the part regarding layout shares essentially the same design in both technologies, so for the following review I will focus on Swing, which has superseded AWT for other reasons that are not pertinent here.

Layout Managers

A *layout manager* is an object that controls the size and position of components inside a container. This concept in essential to meet the platform and device independency: instead of *hard-coding* the layout algorithm in the containers themselves, that knowledge is extracted to another object, in which the container delegates to arrange its children. Java designers have thence applied here the *Strategy* design pattern (Gamma et al., 1995, pp. 315–330), which makes the design flexible and allows interchange layout managers for any container, or even implement our own layout manager, if none of those provided by the Java API serve our needs. An overview of the design of components, containers, and layout managers in Java Swing is provided in the class diagram of figure 6.

These are the layout managers provided by JFC:

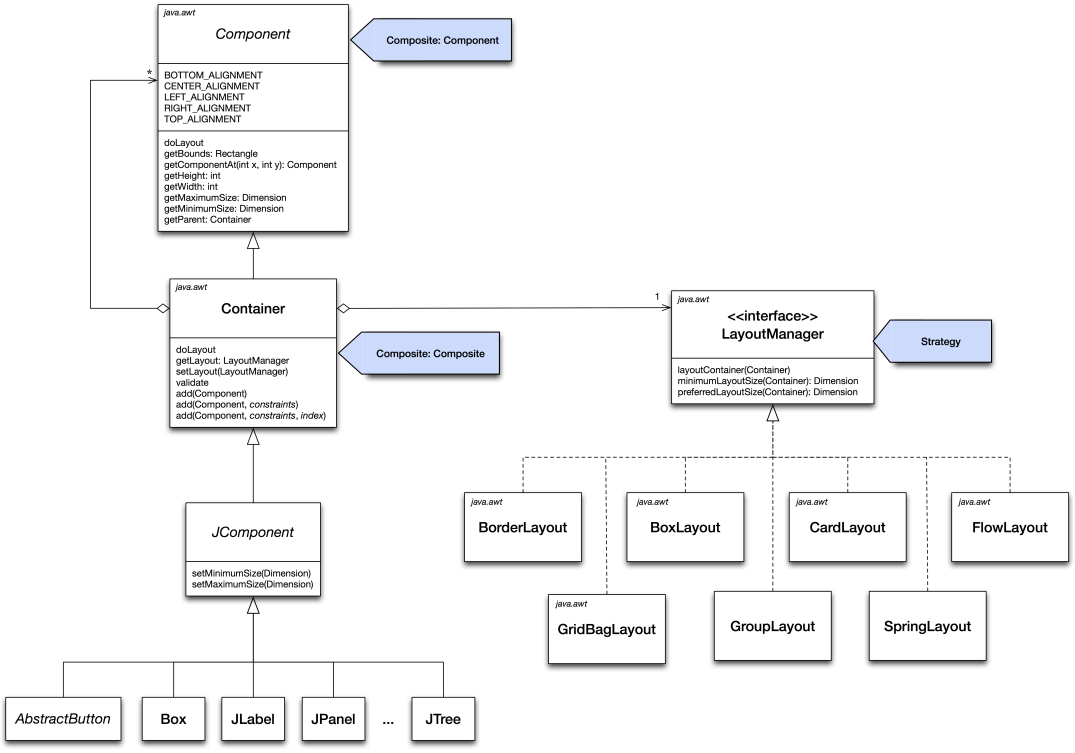


Figure 6. **Class diagram of components, containers, and layout managers in Java Swing. In this flexible design the knowledge of how lay out components is extracted from the container to a class apart, the layout manager, from which the distinct layout policies inherit, thus making very easy to interchange them or even implement our own layout manager. This constitute an example of use of the *Strategy* design pattern. In addition, containers and components follow a *Composite* design pattern, which allows to add either single components or containers to any container without any restriction in the nesting level.**

- `BorderLayout`
- `BoxLayout`
- `CardLayout`
- `FlowLayout`
- `GridBagLayout`
- `GridLayout`
- `GroupLayout`
- `SpringLayout`

There exist, actually, other layout managers, as it can be seen in the Java API documentation for the `LayoutManager` interface (Sun, 2008), where many other classes that implements this interface are mentioned (`ViewportLayout`, `ScrollPaneLayout`, `BasicComboBoxUI.ComboBoxLayoutManager`, `BasicSplitPaneDivider.DividerLayout`, `DefaultMenuLayout`...), but they are very specialised cases of layout managers that are used by the platform itself to perform the operations of some components, or by tool implementors. They are not intended for a general use, as are the ones enumerated above.

I will not go into many details here about them here. Some of them are very similar to those that have already been described for WPF/XAML or XUL. That is the case of `BorderLayout`, which shares similarities with `DockPanel` control in Windows Presentation Foundation (see p. 58), or `BoxLayout` and XUL boxes. It is worth to mention, though, that in the case of the Swing component, `BoxLayout`, it can be combined with the interesting feature of *fillers*, invisible boxes that allow to determine how the excess of space, if any, is distributed among the children of the box. Fillers can be a rigid area, a kind of elastic glue, or even a custom filler created by the developer with whatever *minimum*, *preferred*, and *maximum* sizes (Walrath, Campione, Huml & Zakhour, 2004, p. 355[1]).

GridBagLayout

There is one component, though, that deserves at least a brief mention. It is the `GridBagLayout`. It is the more flexible of Swing layout managers, which " aligns components by placing them within a grid of cells, allowing components to span more than one cell. The rows in the grid can have different heights, and grid columns can have different widths" (Walrath, Campione, Huml & Zakhour, 2004[2]).

1 Also available online at http://java.sun.com/docs/books/tutorial/uiswing/layout/box.html#filler

2 http://java.sun.com/docs/books/tutorial/uiswing/layout/visual.html#gridbag

Discussion

Note that, although it might seem otherwise, I am not pretending to compare XAML or XUL with CSS. At least, not as alternatives. They have very different purposes: both XAML and XUL are user interface languages. They are intended for building graphical user interfaces, in which the content and the layout are a whole. Exactly the same that should be done in the last XUL example to make it looked like the previous one in XAML (adding boxes, altering their order in the markup, etcetera) is the mayor criticism of this thesis to the current way of specifying the layout in CSS. In the case of these languages —and the same happens with Java Swing—, this is not a problem, since *they are not oriented towards the separation of presentation and content*. Although both of them (specially in the case of XUL) allow to separate the style more than I have done in the examples above, in both of them, too, **content and layout are, by definition, inextricably bound**.

What is the point, then, in the comments that I have done during their review, comparing them with CSS? Because creating layouts with them is very easy, when compared with how the same layouts could be done in CSS, using floats and positioning. It is like *drawing*, where a box can be divided into two areas, which in turn are subdivided again, and so on. It is like laying out with Microsoft Word with tables. Or like it was done with HTML table-based layouts. All of those methods have something in common: they are *intuitive*. Unfortunately, not the same can be told of the referred layout mechanisms provided by Cascading Style Sheets. In addition, even some of the very simple layouts that have been shown in this section, are not currently possible in CSS, because the notion of *flexibility* can only be simulated with percentages, and they do not work well when they are mixed with widths or heights expressed in other length units.

The reason, in definitive, why they are being reviewed in this thesis is because some of their features either are related or have served as an inspiration, for the solution presented in this thesis for CSS: the Template Layout Module.

5

CSS Box & Visual Formatting Models

The aim of this chapter is to offer an introduction to the current mechanisms that Cascading Style Sheets offers for layout, namely, floats and absolute positioning. It is a basic review of CSS box and visual formatting models, as they are defined in the current CSS 2.1 specification. Nevertheless, this thesis would not be complete without this review for two reasons: first, it is essential a good knowledge of these mechanisms to understand the more advanced layout techniques examined in the next chapter; secondly, as this chapter will reveal, these basic mechanisms are not so simple after all.

One Document, Two Representations

Both HTML and XHTML are markup languages to describe structured documents, a concept pioneered by the *Standard Generalized Markup Language* (SGML), as it has been explained on *Chapter 2*. While computer encodings of documents have long concentrated on preserving the final form presentation (a nicely laid out paper document), structured document formats take a different approach; rather than preserving the final form presentation they encode the logical structure of the document. Among the reasons for doing so is the preservation of device independence, document searchability and information re-use in general (Lie & Saarela, 1998).

If we focus on the structure itself, it is convenient to see the document as a tree, where every element has exactly one parent, with the exception of the root element, which has none. Thus, the following HTML document could have been represented as the tree shown in the diagram of figure 1 (Lie & Bos, 2005, pp. 28–29).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Bach's home page</title>
  </head>
  <body>
    <h1>Bach's home page</h1>
    <p>Johann Sebastian Bach was a
<strong>prolific</strong> composer. Among his works
are:</p>
    <ul>
      <li>the Goldberg Variations</li>
      <li>the Brandenburg Concertos</li>
      <li>the Christmas Oratorio</li>
    </ul>
  </body>
</html>
```

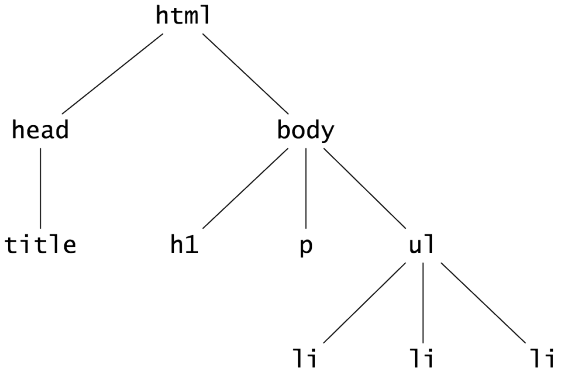


Figure 1. Diagram of elements within elements in a tree structure

The box model defines how user agents process the document tree, generating zero or more boxes for each element.

But this way of viewing documents, though it suits well the notion of HTML pages as structured documents, is not very useful when it comes to actually lay out the document and obtain a visual representation of it. It is there that the box model comes into play, which defines how user agents process the document tree, *generating zero or more boxes for each element*. Thus, instead of representing the document as a tree structure that resembles a genealogical tree —which is very appropriate for focusing on the hierarchical relationships between elements—, now we can see it made up of boxes that contain other boxes. This is what is shown in figure 2, which depicts the boxes that a user agent could have generated for the given portion of a HTML document (Lie & Bos, 2005, p. 123).

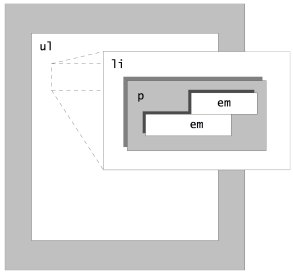
One interesting thing to emphasise about the box model is that, although in CSS it looks as if the style properties are added to elements, what happens in fact is that the browser creates a parallel structure: for each element in the source, an object, called a *formatting object* gets all the properties (Lie & Bos, 2005, p. 124).

Visual Formatting Model Basis

As stated in the specification, when user agents process the document, each element in the document tree generates zero or more boxes according to the box model.

Figure 2. Boxes containing other boxes

```
<ul>
  <li>
    <p>Text of the first in
the list has <em>a few
emphasized words</em> in the
middle.</p>
  </li>
</ul>
```



According to the specification (Bos, Çelik, Hickson & Lie, 2009, §9.1), the layout of these boxes is governed by :

- box dimensions and type
- positioning scheme (normal flow, float, and absolute positioning)
- relationships between elements in the document tree
- external information (for instance, viewport size, intrinsic dimensions of images, etcetera)

What the paragraph above means is that, whereas the box model defines the dimensions of the boxes that are generated from the elements in the document tree, the visual formatting model is at a higher abstraction level and manages how those boxes are laid out on visual media (computer screens, mobile devices, printed pages, etcetera). This is the type of medium with which this thesis is concerned, even though the proposed solution would also bring accessibility improvements to other types of media, like braille devices and speech synthesisers.

In this introductory section about the visual formatting model I will only focus on describing the basic types of boxes which exist in CSS. Box dimensions will be described in the following section. And positioning schemes, which are one of the most important topics of this chapter, will be deferred until the box model and the properties which manage box dimensions have been explained, when they will be thoroughly studied.

Types of Boxes

The visual formatting model defines several types of boxes which may be generated in CSS. The type of a box determines its behaviour, that is, how it is firstly rendered and how it is affected by changes in the viewport dimensions or in the font size.

Block and Inline Boxes

These are the most common types of boxes and, as such, anyone who is taking his first steps in CSS should be aware of the differences between them. Why are them so important? Because they are always present in any HTML document, even in the absence of any style sheet applied to the document. In fact, this distinction is also present in the HTML specification (Raggett, Le Hors & Jacobs, 1999, §7.5.3):

> *Certain HTML elements that may appear in* `BODY` *are said to be "block-level" while others are "inline" (also known as "text level"). The distinction is founded on several notions: …*

The first of such features why is needed to differentiate both types of elements is the *content model*. With this term the specification refers to the fact that the type of an element determines what other elements can contain. Thus, inline level elements can only contain text and other inline elements (note that this include images, since `img` is an inline element), whereas many block elements, like `div` or `blockquote` can also have other block elements inside (nevertheless, it is the HTML grammar, defined by its DTD which has the final word on what concrete elements can be put inside a given one).

Although this distinction is very important from the point of view of the markup, it is not that in which I am interested. Neither is it the third one mentioned in the HTML specification: the *directionality*, which is related to how both types of elements inherit text direction information, a question that is not dealt in this thesis.

What is really important for the topic of this thesis is the notion of *formatting*:

> *By default, block-level elements are formatted differently than inline elements. Generally, block-level elements begin on new lines, inline elements*

Figure 3. Box model basis. The box is actually made up of the borders, the paddings (that is, the separation between the content and the borders of the box), and the content itself. Finally, the margins are the space that a box leaves with the surrounding boxes in the page.

do not. For information about white space, line breaks, and block formatting, please consult the section on text.

Although the specification does not give much details of how both types of elements are formatted, the common behaviour for block elements, since the conception of the web and the first browsers, is that the box generated by them is as width as its containing box, while inline level elements are as narrow at it is possible to fit their contents. This is much more detailed in the CSS specification, and will be explained later in this chapter.

Other Types of Boxes

Block and inline boxes are not, however, the only types of boxes that exist in CSS. Thus, CSS 2.1 defines several values for its `display` property that can make that any element, regardless its type, behaves as if it were a table, a table cell, a list, etcetera.

Box Model Basis

As it has been said, the box model defines how a set of rectangular boxes —those which are going to be actually rendered by the user agent— are generated for the elements in the document. Each box is made up of the following four essential parts (from outermost to innermost): margin, border, padding and content (see figure 3).

Margin

A margin is the space that the bounding box of an element leaves with any other adjacent box on the page. We can think of a margin as having the effect of pushing the element away from other elements on the page (Olsson & O'Brien, 2008, p. 209). Margins are always transparent and, although they belong to the box model, they are actually outside the box.

Border

The border is the outer edge of the box. CSS allows to specify not only the width of the border, but also its colour and style (solid, dashed, dotted, and so on).

Padding

While margins are used to specify the white space around boxes, padding refers to the space inside the box. Or, more specifically, it defines how much space to insert between the content of an element and the outside limits of its bounding box. As such, it is often used to create a *gutter* around content so that it does not appear too close to the outside limits of the box (this is specially true when a background or a border have been applied to the element).

Content

Finally, there is the content itself, which in the CSS box model is the area occupied by the contents of the element for which the box is being generated, either text or other children elements.

Other properties that also affect the box model are:

Width

By default, a block level element generates a box that expands until it takes up all the available width (that is the width of its container box, after subtracting the corresponding paddings, margins and borders), whereas the box for an inline level element is as narrow as possible to display its content without any line break occurs (the actual algorithm for width computation is, of course, rather more

complex). CSS permits to change both the type of box to create and the width of the generated box.

Height

The default height is the height necessary for the content to fit in its box. Of course, that value depends on both the style applied to the content (type and size of font, line height, margins among children elements, etcetera) and *the given width* of the box (the narrower a box be, the taller should have be to fit its contents).

As for the width, height can also be explicitly set, but given the nature of web publishing, this is rarely done currently in CSS, because it depends on several factors over which the designer has not control, and therefore the content may overflow the box (see the note about width and height on the web on below).

Background

Although background properties, unlike the preceding ones, does not change the dimensions of the box, they are also closely related to the box model and affect the visual perception of the generated box.

By default, all elements have a transparent background, which means that the background of the parent box is visible through. But there are some CSS properties that specify either a background image, a background colour, or both.

All the elements of the box model can be seen in 3D, in Hickdesign (2004), of which also exists an interactive version due to Livingstone (n.d.).

A Historical Note about Width and Height on the Web

Since the origins of the web, the default behaviour for rendering HTML documents is that the browser automatically calculates the dimensions of each box based on its contents. The raison d'être for this behaviour is that one of the requirements of the HTML was, says its inventor (Berners-Lee, 1999) "to get it display reasonably on any of a very wide variety of different screens and sizes of papers". Thus, elements in a HTML document with no style applied to

it are laid out according to their type and so that they fit their contents.

Although CSS allows to set an explicit width and height for an element (actually, for the *box* generated by that element), the latter is almost never used on the web for practical reasons, because it is very difficult to determine what is the minimum height of an element for its content not to overlap when either the dimensions of the browser window or the font size change.

Box Dimensions

Once the four distinguishable areas of the box model have been defined, the concrete CSS properties that allow to alter box dimensions and appearance will be described in the following subsections.

Margins

Margins can be set by means of the `margin-top`, `margin-right`, `margin-bottom` and `margin-left` properties, or with the shorthand property `margin`, which allows us to define all the four margin values at once. Margin properties take as a value a CSS length (pixels, points, em[1], centimetres and so on), the keyword `auto` or a percentage of the width of the containing block of the element (if the width of the containing block depends on the element to which percentage margins are applied, the result is undefined in CSS 2.1 Bos, Çelik, Hickson & Lie, 2009, §8.3). These values are enumerated and briefly defined below:

Length

Either an absolute or relative value.

Percentage

A percentage of the width of the block level element that contains this element. This is true even for the top and bottom margins (a percentage still refers to the width of the containing block, not to its height, as it could be thought).

1 An em is a unit of measurement in the field of typography, equal to the point size of the current font (Wikipedia, s.v. "em", http://en.wikipedia.org/wiki/Em_(typography)).

`auto`

The browser will calculate the margins automatically. It is not evident, though, what this means, because it depends on the type of the box generated by the element (whether it is inline, block, inline-block, etcetera), its *positioning scheme* (normal flow, float, or absolute positioning), its width, and the value of the opposite margin. There are too many combinations (Bos, Çelik, Hickson & Lie, 2009, §10.3) as to be reviewed here, but, just as an example of how counter-intuitive it might be, this is the common way of centering block elements in CSS:

```
#menu {
  margin-left: auto;
  margin-right: auto;
}
```

Shorthand Property

In addition to the four individual margin properties (one per each side of the margin area) margins can also be defined with the aforementioned shorthand property `margin`. This becomes a handy tool when we want to apply the same margin to all the four sides of the element, what can be done with just one declaration, like for example: `margin: 2em`. But this shortcut can also be used to specify different values for the four margins. In that case, the values are supposed to be defined clockwise starting from the top. Thus, the following declaration would set a top margin equal to the font size of the element, a right margin of ten per cent the width of the container, a bottom margin of two times the font size of the element and a left margin of 210 pixels:

```
margin: 1em 10% 2em 210px;
```

If less than four values are specified, the missing ones are assigned the same value as their opposite side. So for example a `margin: 1em 2em 3em;` would be equivalent to:

```
margin-top: 1em;
margin-right: 2em;
```

```
margin-bottom: 3em;
margin-left: 2em; /* same as margin-right */
```

Analogous shorthand properties exists for paddings and borders, and they behave the same.

Default Value

The default value for the property `margin` is 0. In practice, though, it is quite common that some elements have an actual value defined by the browser default style sheet. Thus, block level elements usually have some length value applied to their top and bottom margins (that is the reason why in a HTML document with no user style sheet applied to it paragraphs, headings and so on are rendered leaving some vertical space among then). These browser default styles sometimes conflicts with user's intentions, so some authors recommend to override most of such styles to have the style sheet *under control* from the beginning. It is what has been known *reset* stylesheets, which are described later in this chapter.

Negative Margins

As it has been said, margins allow negative values. Although the specification does not define which is their behaviour (at least, *not explicitly*), they have the effect of *increasing the width* of the element to which they are applied. To understand their meaning, consider the following example, inspired on that of Meyer (2007a, pp. 166–167):

```
div { width: 400px; padding: 1.5em 0.5em; }
p.wide { margin-left: 10px; margin-right: -50px; }
```

Paddings

Paddings can be defined with the `padding-top`, `padding-right`, `padding-bottom`, and `padding-left` properties. There exists a `padding` shorthand property, too, which behaves like that described for margins, so it will not be detailed here again. All these five properties accept the following values:

- length
- percentage

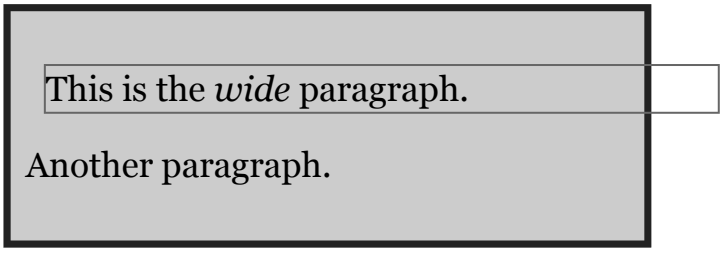


Figure 4. The first paragraph has a left margin of 10 pixels and a margin right of -50 pixels. Note how as a consequence of the negative margin, this paragraph sees how its effective width is increased. In other words, applying a negative margin to one of the sides of an element is like we pull it from that side, *stretching* it.

Their meaning is exactly the same than those for margins, with the only difference that negative values are not allowed for padding.

Borders

Although CSS allows to change the colour, style (solid, dotted, dashed, etcetera), and width of each of the four borders of an element, I will only mention border widths here, since the other properties only affect the appearance of the border, and not the layout (only the width of a border alter the dimensions of the box, as it will be explained soon).

The property that changes the width of a border are `border-top-width`, `border-right-width`, `border-bottom-width`, and `border-left-width`. As always, a shorthand `border-width` is available and lets set the width of the four border sides at a single place.

Width

Finally, once that margins, paddings, and borders have been defined, and their corresponding properties briefly explained, we are able to introduce the `width` property itself. It can take any of the following values:

Length

Any CSS length unit (pixels, em, cm ...).

Percentage

If the width of the element is assigned a percentage value, it has the same meaning than for margins, paddings, and borders: "the percentage is calculated with respect to the width of the generated box's containing block. If the containing block's width depends on this element's width, then the resulting layout is undefined in CSS 2.1." (Bos, Çelik, Hickson & Lie, 2009, §10.2).

But, as usually, things are more complicated in CSS that it seems at first sight. Thus, the `width` property does not set the width of the box itself, *but the width of the content area*. This can be better understood recalling the figure 3, where the four fundamental elements of the box model (margins, borders, paddings, and *content*) were depicted. In other words, the `width` property does not set the width of the visible box compound of border, padding, and content, but only the width of the content. The following formula (Bos, Çelik, Hickson & Lie, 2009, §10.3.3) shows how the width of the box is computed for a "normal" element (a block-level element that is neither floated nor absolute positioned):

> margin-left + border-left-width + padding-left + **width** + padding-right-width + border-right-width = width of the **containing block**

Or, which is the same thing:

> value of **width** property = **available width** - paddings - borders

In order to see how all this elements interact in practice, let's consider the following HTML code:

```
<html>
  <body>
    <h1>The box model</h1>
    <div>
      <p>This is a paragraph inside a
<code>div</code>, which contains text and some
<em>inline</em> elements.</p>
    </div>
</body>
</html>
```

Now I will apply some style to it, focusing on the properties that most directly deal with the box model. For the moment, I will concentrate just in the `div` block to study the basis of the box model and see how the properties that have been reviewed so far in the chapter affect the dimensions of the generated box (later in this chapter I

will have the opportunity to go into deeper details and describe how they interact among them). The applied style is:

```
body { margin: 2em 4em; }
h1   { margin: 0; }

div {
  width: 420px;
  margin: 20px 0;
  padding: 0.6em 10px 160px 30px;
  border: 5px solid rgb(0, 137, 255);
  background: rgb(203, 255, 250) url(wave.png) 35px
bottom no-repeat;
  font-size: 1.2em;
  line-height: 1.5;
}
```

First, a width have been explicitly set for the `div` element. If it had been left unspecified, since `div` is a block element, it would have expanded to occupy all the available width; instead, I have assigned to it an explicit width of 420 pixels to see how that property, `width`, actually works in the box model context.

Secondly, explicit values for margins, paddings and borders are also specified: a value of 20 pixels is set for the four margins; different padding values are also applied to the four sides; and a border of 5 pixels enclosed the whole box. Having all of that into consideration, the width of the box will be **470 pixels**:

box's width = 420px (`width`) + 30px (`padding-left`) + 10px (`padding-right`) + 10px (left and right borders) = 470px

This can be check by means of the Firebug inspector ("Firebug", 2010), a Firefox extension that allows, among other things (for instance, it is also a JavaScript debugger), to inspect the computed CSS for each element, and even provides a kind of "preview" of the boxes generated for the selected element, with its dimensions, in pixels. Thus, the dimensions of the box generated for the `div` block of the example above are shown in figure 5.

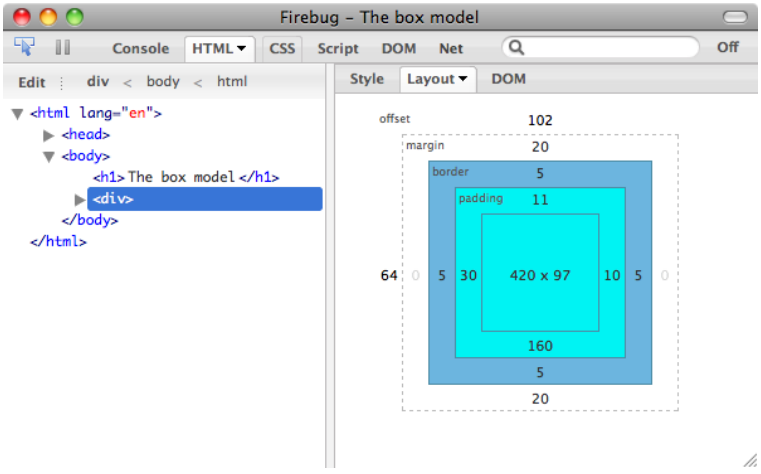


Figure 5. Firebug.

Collapsing Margins

I have postponed the discussion of this issue, instead of mentioning it when margin properties where discussed, because I consider that it is complex enough, and a not so well-known feature of margins, as to deserve its own section. The expression *collapsing margins* (Bos, Çelik, Hickson & Lie, 2009, §8.3.1) "means that adjoining margins (no non-empty content, padding or border areas or clearance separate them) of two or more boxes (which may be next to one another or nested) combine to form a single margin". This only applies to *vertical* margins; horizontal margins *never* collapse.

What are the circumstances under which vertical margins *collapse*? As always, it is not easy to explain, since there are many rules that determine when the margins of an element may collapse with those of its children or surrounding boxes. First, I will define how two margins are combined to form a single margin: *if two or more vertical margins collapse into a single one, the resulting margin is maximum of the adjoining margins*.

As for the needed conditions for two margins to collapse, summarising what the specification says at this respect, for only the most general case, the adjoining margins of block boxes *in the normal flow* (that is, not floated nor absolute positioned) collapse.

There still is another aspect to define: what the term *adjoining* means: two margins are said to be adjoining if *the bottom edge of the bottom margin of the top box and the top edge of the top margin of the bottom element are actually in contact*. For practical purposes, this has two consequences for determining when two margins collapse:

- Whatever adjacent vertical margins collapse, it does not matter whether their boxes are sibling or nested. That is, margin collapsing does not only happen when one block level element follows another, as many people can think (Budd, 2003), but "whenever one margin comes into contact with an adjacent margin. This means that margins can also collapse when one element is contained within another."
- If there is a border between two margins, they are no longer adjacent, and therefore they do not collapse.

I will try to summarise all the above in an example. Let be the following HTML document:

```
<body>
  <h1>Collapsing Margins</h1>
  <p>This is the first paragraph.</p>
  <p>Another paragraph.</p>
  <div>
    <p>This paragraph is enclosed in a div.</p>
  </div>
  <div class="border">
    <p>This paragraph is enclosed in a div that has a
border.</p>
  </div>
</body>
```

Now I am going to apply some margins to the paragraphs and divs:

```
p {
  margin: 20px 0;
  padding: 10px;
}
```

```
div {
  margin: 25px 0;
  padding: 0 1em;
}

div p {
  margin-top: 30px;
}

div.border {
  border: 3px solid white;
}
```

The result is shown in figure 6. Accordant to what has been described about the collapsing of margins, this is what is happening:

- Both first and second paragraphs have their top and bottom margins set to 20 pixels. Since they are adjacent, instead of adding the bottom margin of the first paragraph plus the top margin of the second one, both margins collapse. Both are the same value, so the resultant margin has a value of 20 pixels.
- The third paragraph is enclosed in a div. The style sheet has defined a top and bottom margins of 25 pixels for `div` elements. In addition, the paragraphs inside a `div` are set a top margin of 30 pixels (the bottom one is left unassigned, so it inherits the 20 pixel value from the "normal" paragraphs).

  Now we have three involved margins: the top margins of the div block and the third paragraph, and the bottom margin of the second paragraph: 25, 30, and 20 pixels, respectively. Since the three margins are adjacent, all of them are collapsed, and the result is a margin of 30 pixels. Note that, as has been stated before, there is not matter whether one element is a child of another, or if both are sibling elements: all that counts for margin collapsing is whether they are or not adjacent. This is not usually as intuitive as for the first case.
- Finally, there are the third and four paragraphs, and their containing divs. In this case there are, therefore, four margins that might be collapsed: bottom margins of the third paragraph and its parent div,

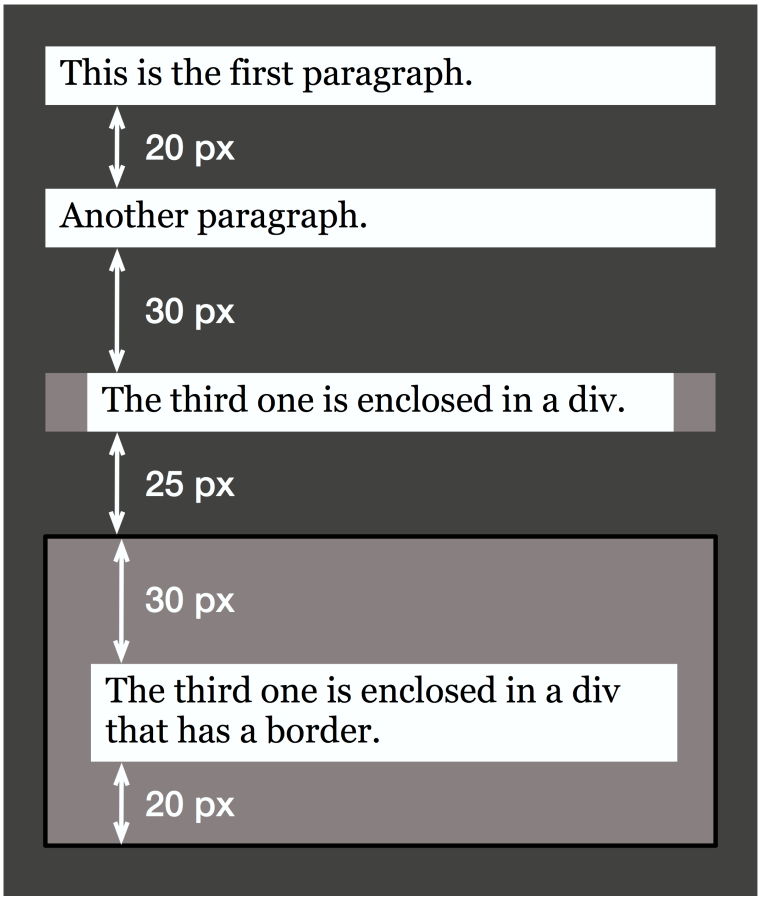


Figure 6. Collapsing margins.

and top margins of the four paragraph and its parent div: 20, 25, 30, and 25 pixels, respectively.

But now another factor comes into play: the div that contains the four paragraph has a border. As stated above, that breaks the adjacency of margins. Thus, the margin top of the four paragraph does not collapse with the others. Instead, it leaves that space (30 pixels) between the paragraph and the top border of its parent. As for the other three margins, the maximum value is 25 pixels: that is the resultant margin after they have collapsed.

Whilst, once learned the theory, it is not difficult to remember, collapsing margins frequently confuse the author. Although in an example like the above, specifically created for illustrating their behaviour, they may even seem, to some extent, *simple*, sometimes, when "debugging" a complex style sheet applied to a real web page, it may be difficult to realise that some unexpected space between two elements is due to this phenomenon. As Meyer (Meyer, 2004a) has stated:

> *Like many basic concepts, margin collapsing can lead to unexpected and sometimes counterintuitive results.*

This is specially true when that unwanted space is due to some nested element. Thus, one of the recurrent "*Why is this happening?*" questions that I am asked in my CSS courses, is, for example, when a student wants a header not to leave space with the edges of the page. They usually have a code similar to this:

```
<body>
  <div id="header">
    <h1>Lorem Ipsum</h1>
    <p id="tagline">...</p>
    ...
  </div>
  ...
</body>
```

And the CSS code that they are trying to apply is something like this:

```
body {
  margin: 0;
  padding: 0;
}
#header { margin: 0; }
```

Certainly, the above would suffice… if it were not for the margins of the `h1` element. The problem is that, most times, as in the above style sheet, they have not assigned any margin to it. Nevertheless,

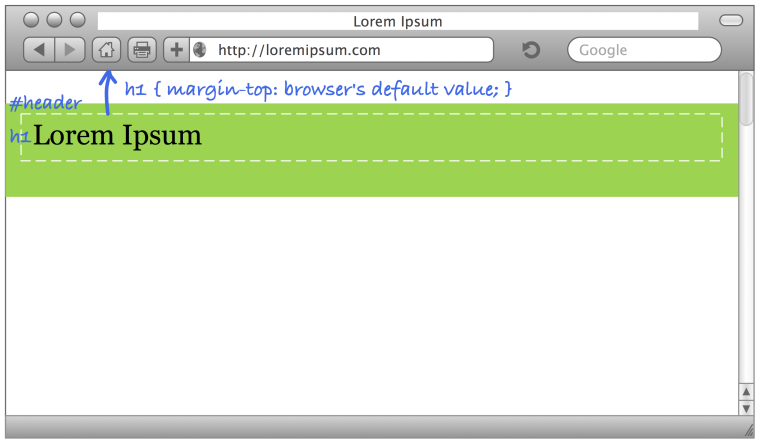


Figure 7. The combination of collapsing margins, nested elements, and browser default styles may lead to unexpected results.

the header is still leaving some whitespace with the top of the `body`, as shown in figure 7.

What is creating that separation, then? In this case, it is due to the default styles that every browser has defined, and which is the cause why, if we open an HTML with no styles applied to it, we are able to see it with a certain visual structure: headings are bigger and bold, lists are formatted as lists, paragraphs are separated each other, etcetera. In the example above, the default margins for `h1` elements is causing the vertical separation between the header and the top of the page. There would have been needed to explicitly remove it with a rule like:

```
h1 { margin: 0; }
```

That is one of the reasons why some authors like to *reset* such default styles, which is known as *reset stylesheets*. This concept is explained in the sidebar of next page.

Floats

According to the CSS 2.1 specification, by floating an element we are shifting it to the left or right on the current line and allowing that the content *flows* along its side (Bos, Çelik, Hickson & Lie, 2009, §9.5). What does it actually mean? As the specification suggests, a

Reset Default Styles

The core idea behind CSS is, as it has been repeatedly mentioned in this thesis, the separation between content and presentation. The content and structure of the document is represented by the markup, whereas all the stylistic information is entrusted to style sheets. Nevertheless, even an HTML document with no style information is not displayed by the browser as if it were a plain text document, but some kind of format is applied to it. Thus, headlines are usually rendered in a larger font size and bold, links are typically underlined and blue, lists are displayed indented and either numbered or with some type of bullet before each list item and so forth. This is because, in addition to style sheets coming from authors and readers, CSS acknowledges a third source of stylistic information, namely the browser. Thus, a browser which supports CSS has a default style sheet that is combined with style sheets coming from the author or reader. Even non CSS compliant browsers, including text browsers, have some hardcoded stylist information.

Although this is intended to be a help for authors, who can then focus just on describing the differences between the conventional presentation (that of the default style sheet) and their preferences, the truth is that it often represents a problem for some users, who sometimes see how the browser is not displaying their web page as it was intended to be. This is specially true in the case of collapsing margins, which have been previously discussed.

In its most basic form, a reset style sheet could be similar to the following:

```
* {
  margin: 0;
  padding: 0;
}
```

In practice, however, authors usually employ a more complex version of it (Çelik, 2004; Yahoo!, 2009; Meyer, 2008a, 2008b, 2007c), where not only margins and paddings are removed, but other styles, such as those of lists, line height, font size, etcetera, are also *reset* to some predictable value, chosen by the author (instead of taking for granted some *hidden* styles chosen by the browser). This is the main reason cited by Meyer (2007b) why he uses a reset style sheet, namely, to *normalise* styles among browser implementations:

> *The basic reason is that all browsers have presentation defaults, but no browsers have the same defaults. … Not only do I want to strip off the padding and margins, but I also want all elements to have a consistent font size, weight, style, and family. Yes, I want to remove the boldfacing from headings and* `strong` *elements; I want to un-italicize* `em` *and* `cite` *elements.*

float (also called *floated* or *floating* box, although the short name is the most widely used in practice) has two effects:

- It moves the floated element to the left or right
- The surrounding content can flow around it

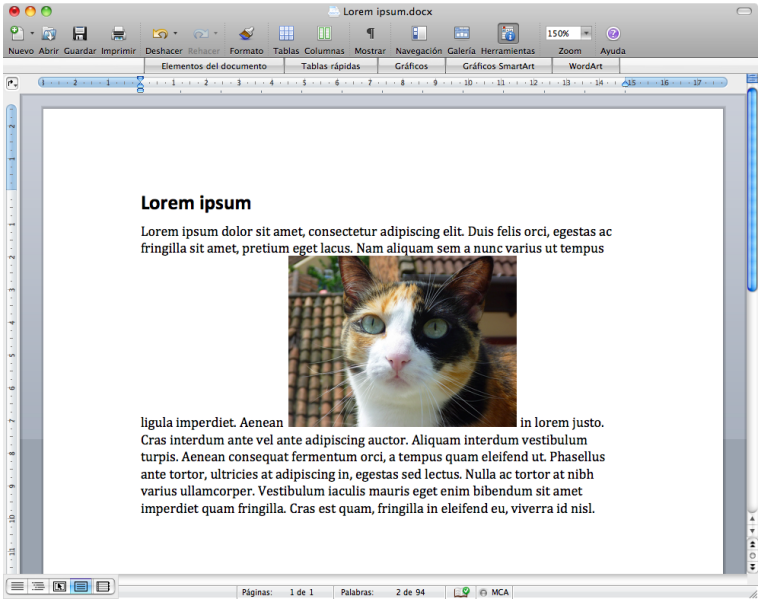


Figure 8. A non wrapping image in Microsoft Word.

By default, inline elements in the normal flow of the document are placed in order, in the same line, one line after another. This is the same behaviour that most desktop word processors exhibits. Figure 8 shows how one of the most popular of such products, Microsoft Word —specifically, its 2008 version for Mac— displays an image that has been inserted in the middle of a paragraph. The same in HTML could have been achieved with the next markup:

```
<h1>Lorem ipsum</h1>
<p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Duis felis orci, egestas ac fringilla sit amet,
pretium eget lacus. Nam aliquam sem a nunc varius ut
tempus ligula imperdiet. Aenean <img src="figures/
lanita.jpg" src="My cat Lana"/> in lorem justo. Cras
interdum ante vel ante adipiscing auctor. Aliquam
interdum vestibulum turpis. Aenean consequat fermentum
orci, a tempus quam eleifend ut. Phasellus ante
tortor, ultricies at adipiscing in, egestas sed
lectus. Nulla ac tortor at nibh varius ullamcorper.
```

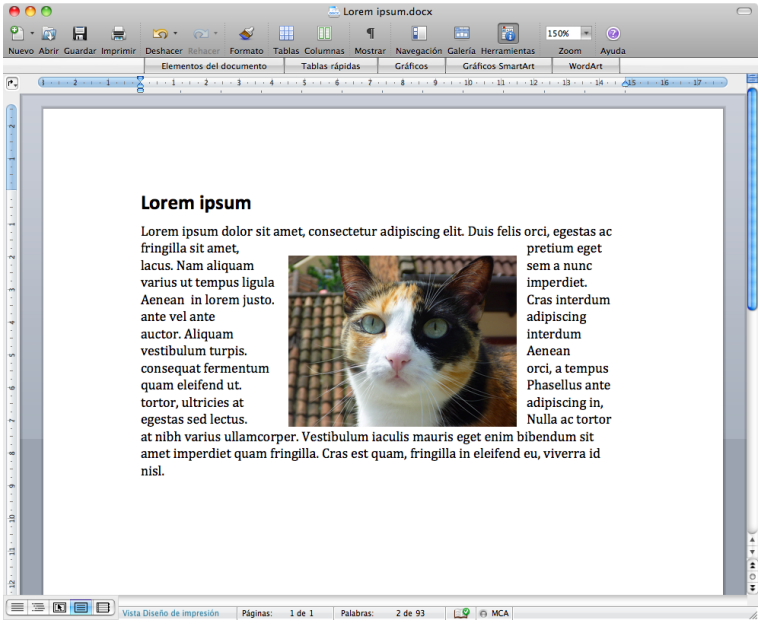


Figure 9. Image wrapping in Microsoft® Word

```
Vestibulum iaculis mauris eget enim bibendum sit amet
imperdiet quam fringilla. Cras est quam, fringilla in
eleifend eu, viverra id nisl.</p>
```

This is not usually what the author wants. What we are used to see in printed media is the text flowing around the images, either on its left, its right or both sides. Again, every word processor allows to do such things, with greater or lesser level of complexity. This can be done in the same word processor of the example above just going to the *Formatting Palette* and then, under *Wrapping* options, selecting a style other than *In line with text* (for example, *Tight*). The result is shown in figure 9.

As we can see, the surrounding text is flowing along both sides of the image. Whereas this is not currently possible in CSS, we do can make the text to flow along either the left or the right side of the image, applying the corresponding value to the property `float`. For example, the next code will produce the result shown in figure 10.

Figure 10. An image floated to the left

```
img {
    float: left;
}
```

Floats were never intended as a layout mechanism, but as a way to let the text flow around the side of a piece of content (usually, an image).

That is for what floats are. This is the main scenario that people who made the first CSS specification had in mind when they included this property in CSS 1 (Lie & Bos , 1996). It is not restricted to images, though, but it can be applied to any element (even an *inline* one, in which case a block box will be generated for it).

Soon, designers started to find other uses of the `float` property. In the beginning, it was no more than subtle variations of the same example described above.

Another common scenario, similar to that of wrapping images that has been shown above, is to float a navigation menu to the left or the right of the page, as for example in figure 11 (a very simple layout that is probably a bit outdated now but was very common a few years ago).

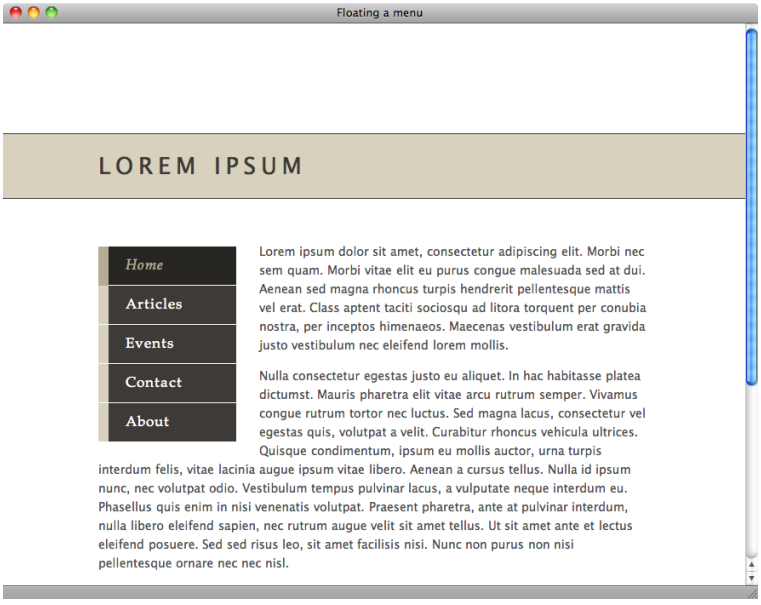


Figure 11. A floating menu

Other Uses of Floats

Other possible uses of floats include:

Anchoring Text

In addition to move the floating element to the left or the right edge of the container, the use use of `float` has another side effect which is useful to get certain effects: although the floated element is taken out from the normal flow of the document, it *remains* relative to the point where the element is present in the document source code, so that it will flow with the text, a feature that can be used to create, for instance, side notes, like those which are found in many books, including this dissertation.

This type of side notes, hanging punctuation, etcetera, can be achieved in CSS using a combination of floats and negative margins:

```
.sidenote {
  float: left;
  margin-left: -12em;
}
```

Drop Caps

A drop cap (also called *versal* or *lettrine*) is a method of marking the start of the text, inherited from ancient scribal practice, which consists on using a large initial capital which expands several lines of text (Bringhurst, 2005). This practice can be recreated in web documents using a combination of `float` property and image replacement, as it can be seen in the example by Weychert (2007) that is shown in figure 13.

The example of drop cap shown in figure 13 can be done with the next CSS code:

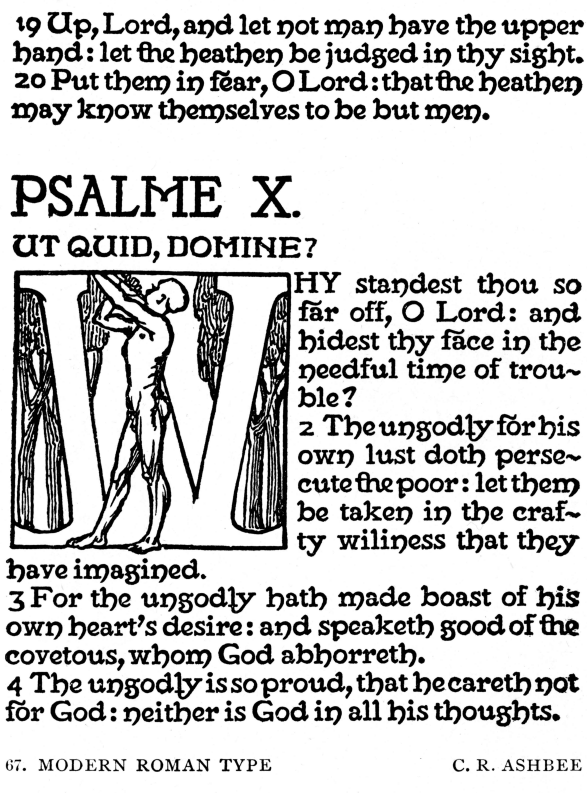
```
.drop {
    width: 83px;
    height: 83px;
    display: block;
    float: left;
    text-indent: -9999px;
    margin: 0 .1em .1em 0;
    background: url(o.gif) no-repeat top left;
}
```

While in the example the author is using an image replacement technique, a drop cap can also be simulated just by enlarging the font size of normal text, as follows, although the result will not be as appealing as the previous one

```
.drop {
  float: left;
  font-size: 6em;
  line-height: .75em;
  margin: 0 .1em .1em 0;
}
```

Float Issues

Even though for these simple use cases it works fairly well, all modern browsers implement it consistently and it is reasonably well understood by users, there are some issues or special features that well worth to mention.



19 Up, Lord, and let not man have the upper hand: let the heathen be judged in thy sight.
20 Put them in fear, O Lord: that the heathen may know themselves to be but men.

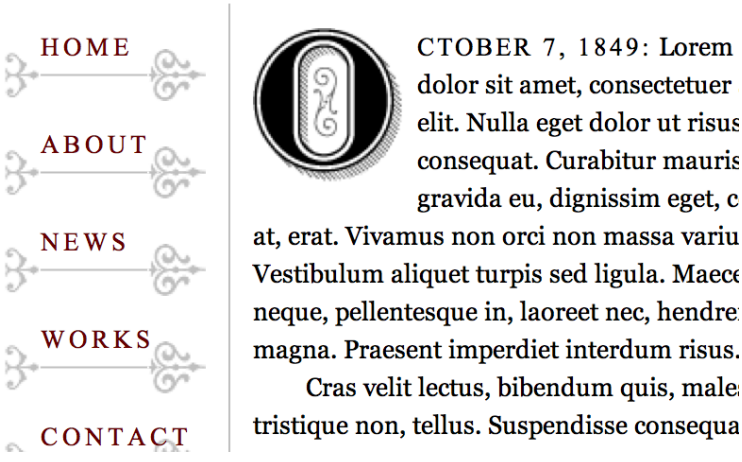
PSALME X.

UT QUID, DOMINE?

WHY standest thou so far off, O Lord: and hidest thy face in the needful time of trouble?
2 The ungodly for his own lust doth persecute the poor: let them be taken in the crafty wiliness that they have imagined.
3 For the ungodly hath made boast of his own heart's desire: and speaketh good of the covetous, whom God abhorreth.
4 The ungodly is so proud, that he careth not for God: neither is God in all his thoughts.

67. MODERN ROMAN TYPE C. R. ASHBEE

Figure 12. An example of drop cap extracted from the book *Letters & Lettering: A Treatise With 200 Examples* (Brown, 1921)



HOME
ABOUT
NEWS
WORKS
CONTACT

OCTOBER 7, 1849: Lorem dolor sit amet, consectetuer elit. Nulla eget dolor ut risus consequat. Curabitur mauris gravida eu, dignissim eget, c at, erat. Vivamus non orci non massa variu Vestibulum aliquet turpis sed ligula. Maece neque, pellentesque in, laoreet nec, hendre magna. Praesent imperdiet interdum risus.

Cras velit lectus, bibendum quis, male tristique non, tellus. Suspendisse consequa

Figure 13. An example of drop cap using CSS by Rob Weychert, in the chapter *Bridging the type divide: Classic typography with CSS* from the book Web standards creativity (Weychert, 2007).

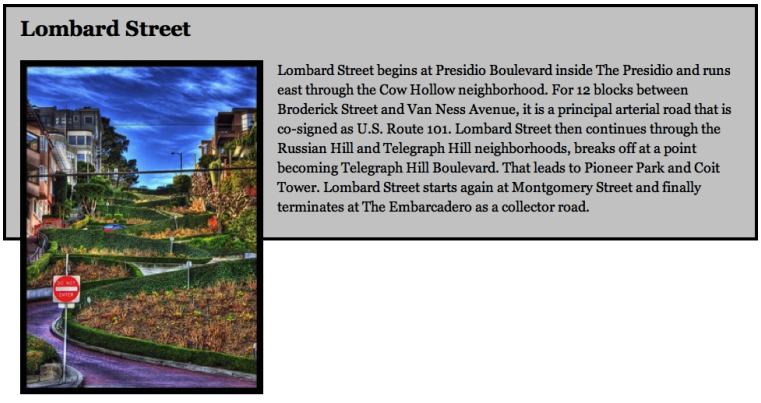


Figure 14. A floated element may overflow its containing block.

A Floated Element Is Taken Out of the Normal Flow

When an element is floated, the box that it generates is no longer in the *normal flow* of the page. However, this is not the same case than for absolute positioned elements. Whereas an absolute positioned element is totally removed from the normal flow, and the browser acts as if such element (its generated box) did not even exist, floats could be said that are in an intermediate position: in Meyer's words (2007a, p. 284), "floated elements exist almost on their own plane, yet they still have influence over the rest of the document".

In effect, we have seen in the examples of this chapter how the browser actually takes care of floated elements, so that the rest of elements in the document do not overlap, but know how to be placed around the float.

However, they are not "normal" elements, either. Specifically, *floated elements are not taken into consideration for computing the height of their containing block* (Bos, Çelik, Hickson & Lie, 2009, §10.6.3):

> *Only children in the normal flow are taken into account (i.e., floating boxes and absolutely positioned boxes are ignored, and relatively positioned boxes are considered without their offset).*

This may led to the situation depicted in figure 14, where the floated image overflows its containing div.

As strange as this behaviour might seem, it is not an error of design in CSS, but an essential feature to be able to float images in

a document, as Meyer (2003b) has explained. But, as Meyer said in the cited article, "while this is necessary for normal text flow, it's a major problem when floats are used for layout purposes".

Clearing Floats

Nevertheless, this issue is not difficult to solve. There are three methods for *clearing* the floats when it is needed that their containing block actually *encloses* them. The best method will depend, as always, on the actual markup of each concrete situation.

Clear Property. The first solution is the obvious one, and which should be applied whenever possible: if the floated element is followed by another one, then it is possible to "clear" it, using the counterpart of the `float` property: `clear`, which prevents an element to flow around the preceding floated elements, placing it in the first line available below then. For instance, to to ensure that `h3` elements are not placed to the right of left-floating elements, the following rule could have been declared:

```
h3 { clear: left; }
```

This property, intended to avoid that floated images hang down into the next section, as in the example above, can also be used to solve the issue of the containing block: if an element with a `clear` property applied to it is in the normal flow of the document, then the containing block will actually be as tall as to contain it and, therefore, the floated element above it.

The problem is when there is no such element to be cleared. One solution would be introduce it artificially in the document. Thus, in the example of figure 14, a new element could have added just before the closing `<div>` tag:

```
<div class="box">
  <h2>Lombard Street</h2>
  <p><img src="lombardstreet.png" alt="Lombard
Street"/> Lombard Street begins at ... </p>
  <p class="clear"></p>
</div>
```

And then cleared with a rule like `.clear { clear: left; }`, or `clear: both;` to be used as a general class for clearing elements, regardless of their position on one side or another.

Of course, this can only be considered a workaround, because by doing so we are introducing presentational markup, thus breaking the separation between presentation and content.

Floating the Container. A second alternative take advantage of an exception to the stated above, regarding the computation of the height of a containing block. While it is true that floated elements, like absolute positioned ones, are not taken into account for the calculation of such height, the specification makes a few exceptions (which, interestingly, are contained in a section titled *Complicated cases*). One of such exceptions is *when the containing block is also floated*. For those cases, the specification states the following for how `auto` heights are computed (Bos, Çelik, Hickson & Lie, 2009, §10.6.7):

> *… In addition, if the element has any floating descendants whose bottom margin edge is below the bottom, then the height is increased to include those edges. Only floats that are children of the element itself or of descendants in the normal flow are taken into account, e.g., floats inside absolutely positioned descendants or other floats are not.*

What does the statement above mean, in plain words? That if the containing block is floated, then it will be as height as needed to enclose all their floated children.

Therefore, in our example, it would be enough to add the following rule:

```
.box {
  float: left;
}
```

Despite its simplicity, this alternative have some drawbacks: it adds more complexity to the layout, introducing a float that, otherwise, would not have been needed. While this has not necessarily to be a problem (assuming that browser vendors implemented CSS 2.1 *perfectly*), it might sometimes interfere with other floated elements in a complex layout. In addition, by making the container floated,

we have only moved the problem a step upwards: now, it is possible that the container of the container only have floated elements, and therefore it would have to be "cleared", and so on.

Using the Overflow Property. According to Koch (2005), the third alternative for "autoclearing" floats here presented was first written by Walker (2005), who in turn give the credits for its invention to O'Brien (n.d.). The method consists on defining the following two declarations for the containing block:

```
.box {
  overflow: auto;
  width: 100%;
}
```

However, this technique is probably the most problematic of all the reviewed ones. Specifically, as Knoch (2005) acknowledges, it might cause unwanted scrollbars under certain circumstances. Although the method also works specifying `hidden` as the `overflow` value, it is even worse, because in such case the content would may be hidden.
Nevertheless, Clarke (2007a, p. 307), talking about the ways of clearing floats without added markup, has said: "one of the simplest, and my current preferred solution, is to use the `overflow` property".

Generated Content. The last method for clearing floats is based on the generating content mechanisms provided by CSS (Bos, Çelik, Hickson & Lie, 2009, Chapter 12). It is based on the first reviewed method but, instead of inserting a presentational element into the markup, it is generated from CSS. The solution is as follows:

```
.box {
  content: '.';
  display: block;
  clear: both;
  visibility: hidden;
  height: 0;
}
```

What Flows Is the Content, Not the Box Itself

Other float issue, specially when it is being used for layout purposes, is what I have tried to summarise in the title of this subsection: *is the content that flows, instead of the box itself*. Or, more precisely —although probably less understandable too— (Bos, Çelik, Hickson & Lie, 2009, §9.5):

> *Since a float is not in the flow, non-positioned block boxes created before and after the float box flow vertically as if the float didn't exist. However, line boxes created next to the float are shortened to make room for the margin box of the float. If a shortened line box is too small to contain any further content, then it is shifted downward until either it fits or there are no more floats present.*

What the paragraph above means is that are not the block boxes themselves, but the lines inside them, which flow down along the opposite side to that where the floated element is positioned. This can be easily illustrated setting a background colour or a border to the element next to the float. In this case, that background colour will be shown under the floated element (as long as we left the background colour of the floated element as transparent). This is a not so well-known feature of floats, as I have been able to verify in the CSS courses that I teach, and can sometimes be an issue when floats are used for layout.

6

CSS Layout Techniques

Whereas last chapter has described the basic layout mechanisms provided by CSS , this one reviews the state of the art of how such mechanisms are being by web designers, in an attempt to overcome their limitations.

The chapter begins defining the different types of layout on the web, their advantages and inconveniences. Then, some advanced techniques, like creating several columns in any order or how to simulate equal-height columns are discussed.

Introduction

Whereas the hypothesis of this thesis is —expressed in a very succinctly manner— that CSS suffers from a lack of true layout capabilities, the truth is that it seems to be an easily arguable sentence when one looks at the designs which can be seen on the web nowadays. Headers, footers, two and three columns layouts, etcetera, are things that are found in almost every current web site. Does it mean that the hypothesis is false? No, *it does not*. While almost any imaginable design can be achieved using just standard (X)HTML and CSS, it is also true that many times they are carried out pushing CSS beyond its limits, using properties in a manner for which they were not conceived, or resorting to tricks that recall those used in the early stages of the web.

Types of Layouts

This introductory section defines the different types of layouts that can be currently found on the web.

As it has already been mentioned in this thesis (see p. 24 on *Chapter 2*), one of the major differences between the web and other media is the lack of control that web designers have over the dimensions of the user's browser window or the font size being used, contrasting with traditional printed-media design. Although they are not the only factors that may vary (installed fonts is, for example, another common issue in web design), both resolution and font size are the two variables that exercise the most influence over the layout.

This section reviews the different type of layouts based on how they adapt to changes in either the resolution of the browser window or the font size, analysing their historical trends on web design and stressing the advantages and drawbacks of each technique.

Fixed Layouts

Fixed layouts are so called because the designer sets the width of the overall layout of the page to a certain value in pixels. Their main characteristic is that the width never changes, regardless the user's screen resolution, the dimensions of the browser window[1], or if he increases or reduces the font size through the preferences of his browser. That is the reason why they are also known as *rigid* layouts.

They have been considered wrong for a long time, because of their inability to adapt to the specific user's environment, which has been traditionally seen as an accessibility issue (Allsopp, 2000):

> *Browser windows can be resized, thereby changing the page size. Different web devices (web TV, high resolution monitors, PDAs) have different minimum and maximum window sizes. As with fixed font sizes, fixed page layout can lead to accessibility problems on the web.*

In a more recent article, Nielsen (2006a) also advocates for using liquid layouts:

> ***Use a liquid layout*** *that stretches to the current user's window size (that is, avoid frozen layouts that are always the same size).*

(Even thought then he nuances the sentence above saying that the design *should be optimised for 1024 × 768*, but using a liquid layout that stretches well for any resolution, from 800 × 600 to 1280 × 1024[2].)

Nevertheless, in the last years, some authors have rebelled against this belief. Thus, it was the subject of many debates (Whitespace, 2003 ; Mullenweg, 2003; Marcotte, 2003; Keith,

1 Note that browser window size is not the same as screen resolution: not all users browse the web with their browser window maximised (although it is true that so do a majority). According to Baekdal (2006), 20 percent of the users with 1024 × 768 resolutions does not maximise their browser window, so to support 95 percent of the users, authors must design for a maximum size of 776 × 424 pixels, despite he found in the same report that only five per cent of the users had 800 × 600 screens.

2 Of course, concrete resolutions will vary over time, depending of users' screen sizes and browsing habits, but the essential idea of Nielsen's article is that the design must be optimised for the commonest resolution yet supporting other typical resolutions, both larger and smaller, and this can only be done by using liquid layouts instead of fixed ones.

2003; Rutter, 2003) in the web standards community when two renowned designers such as Douglas Bowman and Dan Cederholm coincidently (and, according to them, *coincidentally*) changed their personal sites, *Stopdesign*[1] and *SimpleBits*[2], respectively, from a liquid layout to a fixed-width design, on December 2003.

Bowman (2003) explains that such switching was primarily due to two reasons:

The difficulties for controlling the length of lines

A basic typographic rule sets that more than 75 or 80 characters per line of text becomes difficult to read, being the 66-character line (counting both letters and spaces) widely regarded as ideal (Bringhurst, 2005, p. 26). Fluid layouts that expand all the available width may go far beyond that limit in large screens, thus affecting the legibility of the page.

Images and other fixed-width objects

It is difficult to insert a fixed-width object, such as an image or a video, when the width of the container is not known (as in a liquid column) while preventing content overlapping and preserving the layout.

Whereas the line-length issue could have been solved (leaving aside the lack of support in some browsers) with the `max-width` property, setting a value in *em*, the issue with replaced elements (those that have their dimensions defined by an external source, such as images) and fluid layouts is still a problem nowadays (notwithstanding some modern techniques that have appeared to deal with it, which are discussed later in this chapter).

Another example of fixed layout can be found on *A List Apart*[3]. When this acclaimed online magazine devoted to evangelise about web standards was redesigned in 2005, their authors not only decided to use a fixed layout, but they also optimised it for a width of

1 http://stopdesign.com/
2 http://simplebits.com/
3 http://www.alistapart.com/

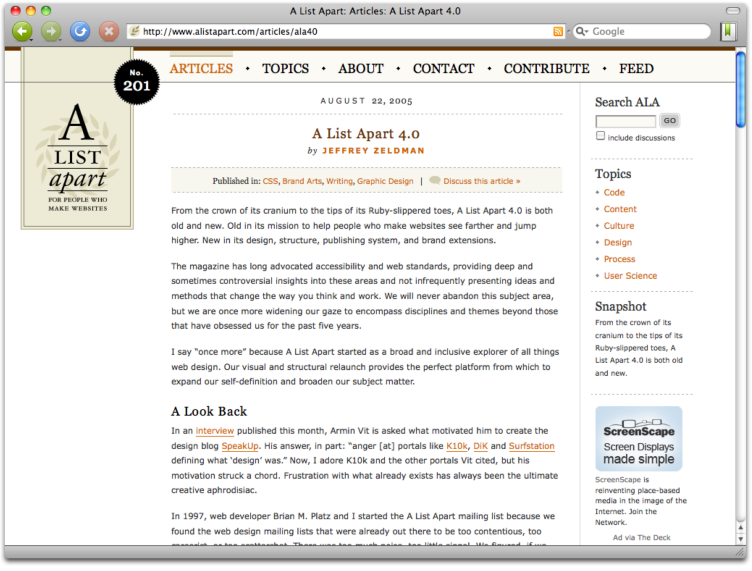


Figure 1. An example of fixed layout, *A List Apart*. When this online magazine about web standards was redesigned on August 2005, their creators not only adopted a fixed layout —against the common trend then on web standards community of fluid layouts—, but they did it for a resolution of 1024 × 768 pixels, when almost everybody were designing for 800 × 600.

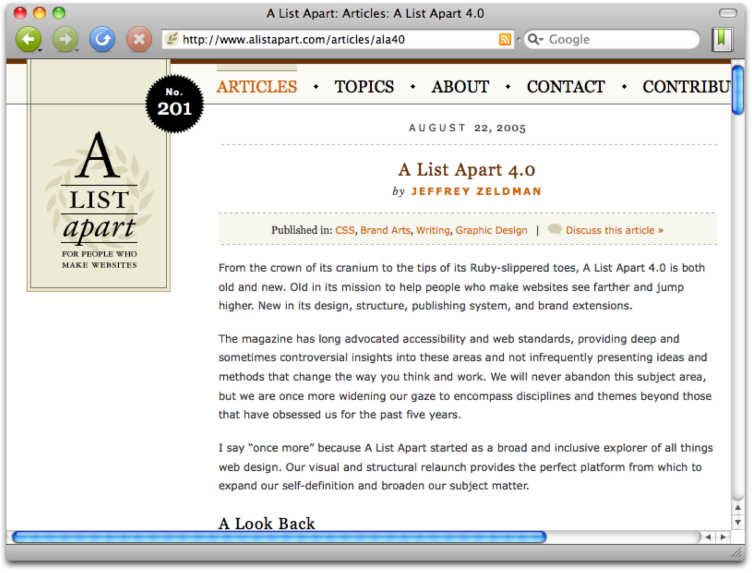


Figure 2. When viewed with a browser window size of 800 × 600 pixels, the layout of *A List Apart* does not change, but remains at a width of 1024 pixels, so that horizontal scrolling is needed to see all the contents of the page.

1024 pixels (when most fixed designs were made for the more conservative resolution of 800 × 600 pixels). The result of such redesign is shown in figure 1. When it is seen at a 800 × 600 resolution some of the contents of the page are no longer visible, and a horizontal scroll bar appears at the bottom of the window (figure 2).

Being asked about this question, Santa Maria (2005), one of the renowded designers in charge for such redesign, said:

> *ALA [A List Apart] has always tried to be one of those sites at the front of the pack. We don't support 800 × 600 anymore, nor do we 640 × 480. Do you? People flipped when sites stopped supporting 640 × 480… now no one says a word. Things change. Trust me, you are going to see more sites stretching out their legs and putting their feet up.*

What resolution?

A fundamental question to be considered when designing a fixed layout is to decide the resolution for which it will be optimised. Unfortunately, it is not easy to know a priori what it should be, because, despite there are some studies about screen resolutions, as it has been aforementioned in this chapter, *screen resolution is not the same as browser-window size* (see footnote 1 on page 107), and the latter is much more difficult to find out.

What is important to note is that a fixed layout is always a tradeoff: a too conservative approach, as for instance designing for a resolution of 640 × 480 pixels, will probably annoy most users, whereas if the design is optimised for, let us say, a width of 1280 pixels, many of them who do not maximise their browser window will have to make horizontal scroll to be able to see all the contents of the page, regardless of their screen resolution.

An option may be to follow current trends on the web. For instance, the table on figure 3 shows the results of a comparison made by Tanaka (2009) of the evolution of the layout width of some large web sites between 2006 and 2008. While the four compared sites were optimised for a width of 800 pixels in 2006, in 2008 all of them have switched to 1024-width layouts.

Mobile Devices

But, if, in the past, opting for a fixed-width layout was to a great extent a matter of choosing the appropriate resolution for which it was to be optimised, the growth of mobile devices has only made

	2006	2007	2008
Apple	800 × 600	800 × 600	1024 × 768
Microsoft	800 × 600	1024 × 768	1024 × 768
Yahoo!	800 × 600	800 × 600	1024 × 768
Youtube	800 × 600	1024 × 768	1024 × 768

Figure 3. Evolution of the layout width of some large web sites between 2006 and 2008 (Tanaka, 2009).

things more complicated. The horizontal scrolling issue[1] inherent to this type of layouts becomes much more noticeable (and problematic) when the small screens of these devices come into play. Even with the sophisticated browsers that these devices incorporate nowadays, and their *adaptation* algorithms and *page zooming* capabilities, browsing a fixed-width page is often frustrating, due to the combination of both horizontal and vertical scrolling that is needed to read a web page.

It can be argued that liquid layouts are not the right solution either. Despite they tend to adapt, by nature, much better to small screens than fixed-with layouts, the browsing experience is so different in a mobile device and the difference between the resolution of computer screens and mobile devices so big that it is very difficult to create a design that works well for both usage scenarios. Even an alternate stylesheet is not usually the definitive answer to this issue (although it is better than simply do nothing, of course), but, as far as far as usability is concerned, the optimal solution necessarily involves creating another version of the web site (Nielsen, 2009a; 2009b; 2006a). The problem is that not every site can afford the costs of maintaining a completely different version of the site. And, in that case, what should be the supported devices? New devices are constantly appearing: in addition to mobile phones, other devices

1 Actually, there is nothing inherently wrong with horizontal scrolling itself. As Braganza et al. (2009) found in an experiment conducted to compare different layout and scrolling strategies for the same document that participants spent less time scrolling and scrolled less often with horizontal-scroll layout. The problem is when users have to scroll *in both* directions.

such as book readers, portable consoles, notebooks and tables are becoming more and more popular, as is their use for browsing the web (and this trend can not but increase), each having very different screen sizes.

Thus, despite from an usability point of view, the *ideal* solution is having separate versions of the site optimised for the resolutions and interaction features of these devices, it is also true that for those who can not afford to maintain several versions of the site, as a general rule a liquid layout will degrade better when it is viewed on a small screen.

Liquid Layouts

Liquid, or *fluid* layouts, are those whose width dynamically changes to adapt to that of the user's browser window: whenever the user increases or reduces the width of his browser window, so does the layout stretch or shrink accordingly to fit all the available width (or a percentage of it).

Liquid layouts can be done in CSS setting the width (of the whole layout, its columns, or both) in *percentages*. For instance, a two-column layout that fit the whole window can be done with the following rules:

```
#nav {
  width: 33%;
}

#maintext {
  width: 67%;
  float: right;
}
```

The result of the rules above is sketched on figure 4, which shows how the layout fit all the available width of the browser window (since the sum of the widths of the two columns is 100%) and the whole layout adapts proportionally when the dimensions of the browser window change.

Issues with liquid layouts

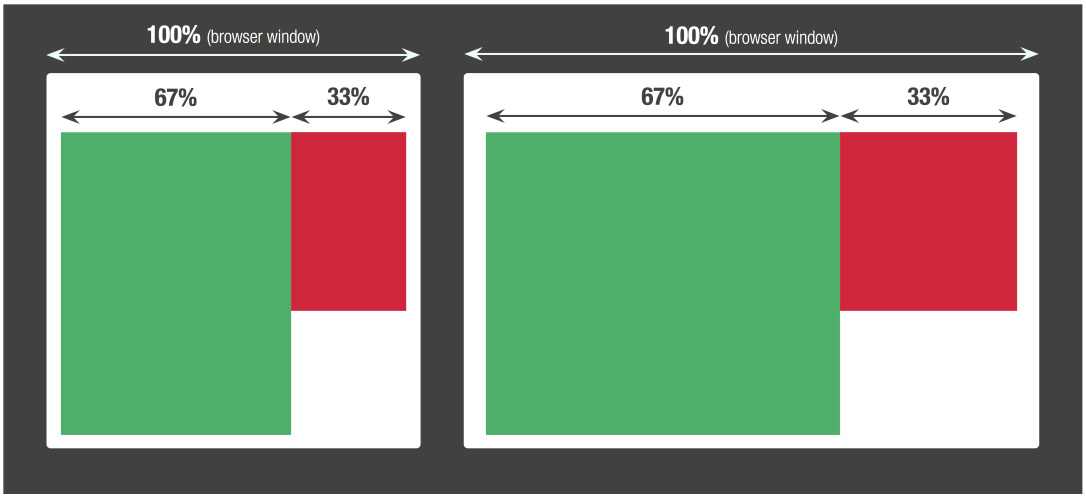


Figure 4. A liquid layout made up of two columns. The main content area (on green in the figure) has a width of 67%, and the navigation menu (on red) has a width of 33%. Therefore, both columns will always fit all the available width of the browser window, stretching and shrinking as it is enlarged or reduced.

Although this ability to accommodate to the dimensions of the browser window seems to indicate that liquid layouts are better than fixed ones, they run into other problems, the *length of text lines* being the most frequently cited.

Another issue of liquid layouts, already commented on page 108 is that of images. When images and other fixed-width elements are inserted into a liquid layout, they do not mix well with the proportions.

All of that has led Zeldman (2003, p. 278) to say:

> *The CSS box model works well when used to create layouts based on absolute pixel values, but its complexity can make it cumbersome, unintuitive, and even counterproductive when you try to create percentage-based ("liquid") layouts that reflow to fit the visitor's monitor.*

Elastic Layouts

Another possibility that tries to conciliate the flexibility of liquid layouts with the control over the line length of fixed-width designs is what has been known as *elastic* layouts. In them, the width of the

columns are expressed in *em* units, so that they are resized when the user increases or reduces the size of the font, while maintaining the readability.

Hybrid Layouts

Finally, it is possible to create countless variations of the mentioned types of layouts simply by mixing different units of measurement or limiting the flexibility range of a liquid or elastic layout (Gillenwater, 2009, p. 11).

The Long Debate

As Clarke (2007a, p. 90) has stated, "arguments will no doubt continue to rage on between those who prefer to make fixed-width designs for what they think is the average minimum window width (nominally 800 pixels) and those who believe designs should be flexible to make the most of a visitor's screen space, no matter what the window size". The same idea had been pointed out by Holzschlag (Holzschlag, 2005b), talking about the limitations of CSS to manage line-lengths:

> *It is my belief that this limitation has frustrated Web designers significantly enough so as to cause a desire to regain at least some of the width control we had when we used centered, table-based layouts. This fixed-width type of layout was extremely prevalent for a time, until a move toward liquid layouts became all the rage.*
>
> *But sometime in the past two years, the reappearance of centered, fixed designs emerged —and from an intriguing place: the Web standards design community. From Douglas Bowman to Jeffrey Zeldman and right on down the line, fixed, centered designs have stolen the day. An interesting phenomenon indeed.*

Although such debate is still open, and it is out of the scope of this thesis to incline for one or another layout strategy, this section has shown the different available options.

Equal-Height Columns

One of the most obvious flaws of CSS concerning layout is its inability to define two elements of the same height, something that is an essential feature to keep a sense of vertical motion in any layout, and that is absolutely necessary to obtain two or more equal-height columns. Of course, it is possible to define an explicit height for the involved elements, using any CSS length unit or a percentage, as for example:

```
#content, #sidebar {
  height: 600px;
}
```

But, as it has been stated in the previous chapter, setting explicit heights in CSS is not practical in most situations, because the self nature of the web keeps us from knowing which is the appropriate height that an element must have to confine its contents without overlapping or cropping issues. Only in certain scenarios, for specific blocks of content that the designer have under his control, may be possible to define a height in "em" and yet be confident, to a certain extent, that the so defined columns will remain the same height when the resolution or the font size change. This is what Clarke (2007b) does in the example shown in figure 5. The code responsible for making the sections *Worriers*, *Worries* and *Worrydone* of the same height is excerpted below (Clarke, 2007b, p. 93):

```
#worriers, #worries, #worrydone {
  float: left;
  min-height: 42em;
  width: 220px;
  margin-right: 20px;
  padding-bottom: 1em;
  background: url(../images/worryone-b.png) no-repeat
              0 100%;
}
```

Figure 5. In this example, Clarke (2007b) is using an explicit height (`min-height` to be precise) to force *Worriers*, *Worries* and *Worrydone* sections to have the same height, without adding any container (as most techniques for simulating equal-height columns do). This approach is error prone and can only be applied in very concrete situations. In addition, it requires that the elements for which an explicit height is defined have also a fixed width.

Faux Columns

Some techniques have been developed to overcome the inability of CSS to create two or more columns of the same height, of which the most popular is *Faux Columns* (Cederholm, 2004).

Actually, this technique does not make the columns equal in height, but creates that *illusion* applying a vertically tiled background image to their container (depending on the complexity of the layout and the design of the site, sometimes a background colour may suffice).

In his article, Cederholm (2004) explains how this technique is applied to his own web site, *Simplebits*[1], which is shown in figure 6 as it could be seen on January 2004, when his article was published.

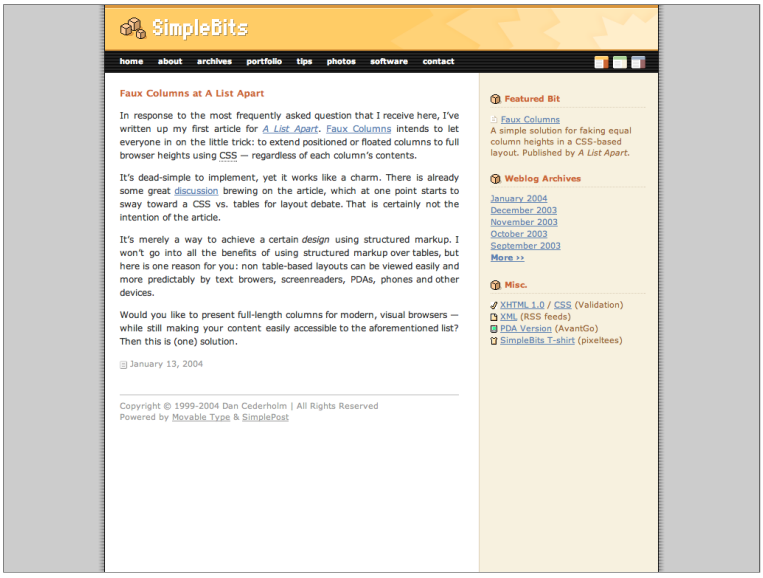


Figure 6. Simplebits, the web site of Dan Cederholm, author of *Faux Columns* technique, as it could be seen on January 2004.

As it can be seen on that figure, *Simplebits* is using a two-column layout. If we open it in a web browser, we will discover that it has a fixed layout (that is, it does not fit the size of the browser window), centered on the screen. Examining its style sheet, we find that it has a width of 750 pixels, with a right sidebar of 240 pixels, being the remaining width (750 - 240 = 510 pixels) for the main content area of the page. As it can be seen, both columns are —or, at least, look as if they were— the same height. To accomplish such design, Cederholm is using a background image (see figure 7) that "is no more than a few pixels tall, but when vertically tiled, it creates the coloured columns that will flow all the way down to the bottom of the page —*regardless of the length of content in the columns*" (Cederholm, 2004).

The background image is being added to the `body` element with the next declaration:

1 simplebits.com is the web site of Dan Cederholm, author of *Faux Columns*. Although the design of the site has changed, the version that is described here, corresponding to January 2004, is available at http://web.archive.org/web/20040117030932/http://www.simplebits.com/index.html.

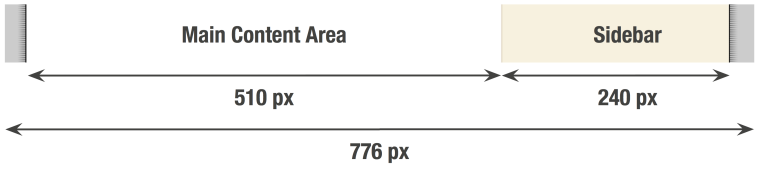


Figure 7. The background image used in *Simplebits* for applying the *Faux Columns* technique. When it is vertically tiled, it creates the visual effect of having two equal-height columns: a blank area for the main content of the page and a coloured sidebar on the right. In addition, decorative borders are added to both sides of the layout, that is conceived to be centered on the browser window when it is viewed at resolutions greater than 800 × 600 pixels.

```
background: #ccc url(../images/bg_birch_800.gif)
repeat-y 50% 0;
```

The background image is horizontally centered on the `body` element (thanks to the `50%` value, a shorthand for `background-position: 50% 0`), to guarantee that it matches the contents of the page, also centered on the screen. Note that this technique does not deal with how the columns are actually laid out, that is, whether absolute positioning, floats or whatever other method is being used for the layout of the columns. It focuses on giving them the appearance of being the same height. As Cederholm states, although *Simplebits* is using absolute positioning to create the two-column layout, "equally fine results can be achieved via the `float` property".

The technique is so simple that it is not surprising that similar approaches had already been used prior its publication. Thus, several months before, the Salmon Cream Cheese[1] submission to *CSS Zen Garden* (Shea, 2003) already used the same concept to obtain the design shown in figure 8. The underlying structure of how that layout is done is in turn shown in figure 9. As it can be seen in both figures, such design consists of a main content area that fits the browser size (a *liquid* column) plus a left sidebar for the secondary content ('select a design', 'archives' and 'resources'), which has a *fixed* width of 224px. Both columns expand vertically the whole length of the page. Although in this case it is not an image, but a plain colour, what is applied to the container (which, again, is the `body` element), the underlying concept is the same than that of *Faux Columns*. Cederholm's merit is to have been the first person who documented this technique, gave it a name, and demonstrated how

1 Shea, D. (2003, May 7). Salmon Cream Cheese. Retrieved from http://www.csszengarden.com/?cssfile=002/002.css

Figure 8. Dave Shea's *Salmon Cream Cheese* submission to *CSS Zen Garden*. Despite it had been submitted several months before the publication of Faux Columns by Dan Cederholm (2004), it is using a very similar method to achieve coloured columns that seem to be the same height. The only difference with Cederholm's technique is that, in this case, a simple background colour is enough to obtain the desired effect, without the need of a background image.

an image could be applied to achieve more sophisticated effects, like adorned borders.

More Columns

Although the two examples described above are two-column layouts, the same technique is still applicable for whatever number of columns. As long as the width of each column is defined in pixels, it is simply a matter of creating the appropriate image using some graphical editor. As it has been shown, it is even possible to have one liquid column, although it must be the rightmost or leftmost one. If that is not the case, it is even possible to use this technique, at the cost of adding extra containers. But this would only address the problem of simulating they are the same height: it is still needed to work out the issue of the layout itself. For instance, how could it be

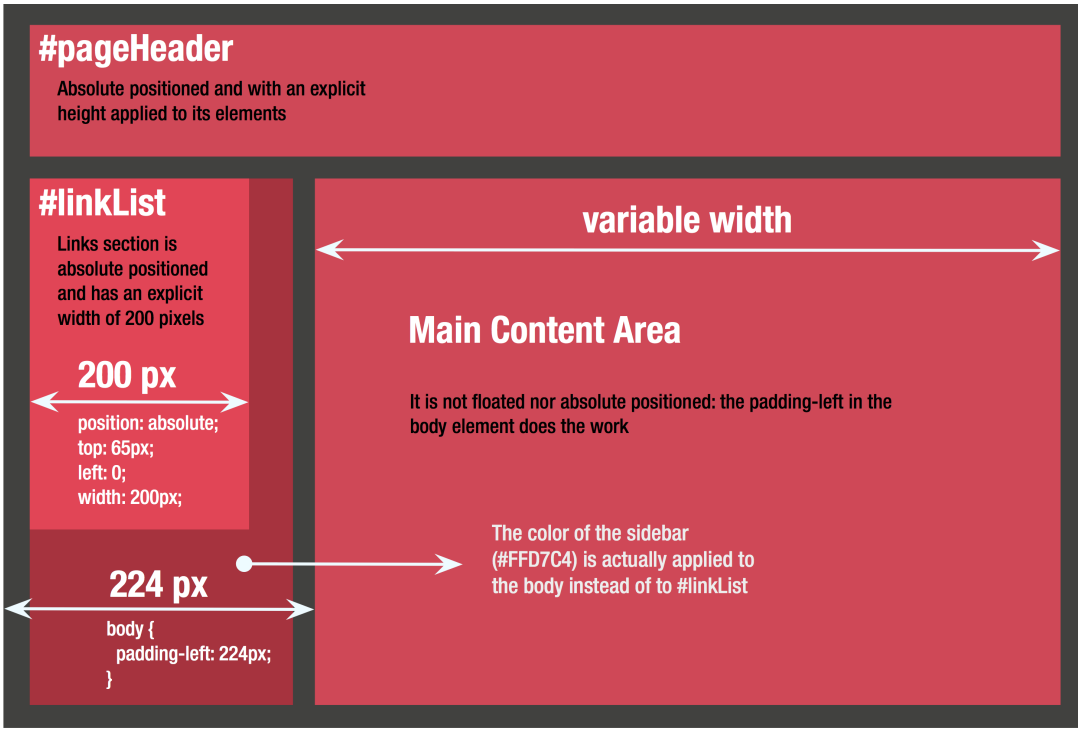


Figure 9. Visual structure of *Salmon Cream Cheese*, a Dave Shea's design submitted to *CSS Zen Garden* several months before Faux Columns were published. In this case, a solid colour is applied to the background of the body element, instead of an image, but the underlying principle is the same. As in *Simplebits*, absolute positioning is being used for the layout of the columns, although in both cases floats would have been equally valid.

done a three-column layout where the column in the middle is liquid?

Sliding Faux Columns

Both examples described above are using either a fixed layout or, in the case of *Salmon Cream Cheese*, an *hybrid* layout in which one of the columns is fixed. But it does not mean that this technique is not feasible for *pure* liquid designs (that is, those in which the width of every column is defined in percentages).

Although the original *Faux Columns* technique focuses only on fixed-width layouts, the same idea of using a tiled background image can be take a step further, as Bowman (Bowman, 2004) and Meyer (Meyer, 2004b) did, developing the *Sliding Faux Columns* approach,

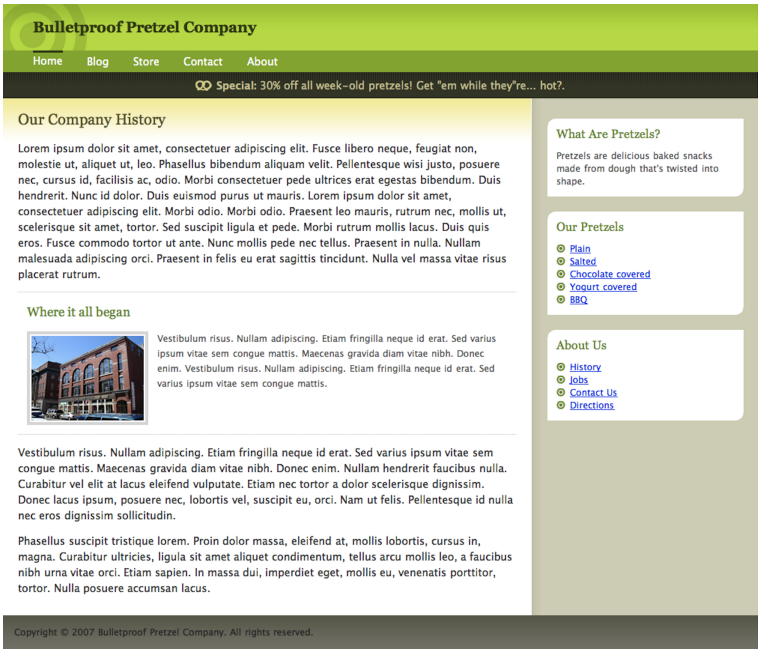


Figure 10. A liquid layout with two columns of 70% and 30% in width, respectively (Cederholm, 2008, p. 249).

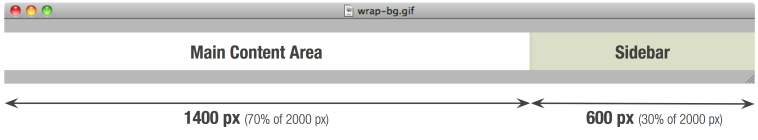


Figure 11. The background image for *Bulletproof Pretzel Company* is 2000 pixels wide (an arbitrary width chosen to span all the design even when it is viewed on large screens). By specifying the areas corresponding to each column in the layout proportionally wide to the width of each column, we will then be able

"where the tiled image can *slide* around behind fluid-width columns, thus creating the equal-height effect while remaining flexible[1]" (Cederholm, 2008, p. 226).

To understand how it works, let us suppose a two-column liquid layout, like that created by Cederholm (2008, pp. 249–285) that is shown in figure 10. The web site of the fictitious *Bulletproof Pretzel Company* specify the columns to be 70% and 30% in width, respectively. The basis of this variant of the *Faux Columns* technique consists

1 Note that Cederholm (2008) is using here the term *flexible* as a synonym of *liquid*, or *fluid*, layouts that were described in the opening section of this chapter, where the different types of layouts that can be done with CSS were enumerated and briefly discussed.

on creating the background image wide enough to accommodate large screens; for instance, 2000 pixels. Then, the area of the image corresponding to each column in the layout must have a width proportional to its column. In this case, the background colour with a shadowed edge for the right sidebar must be 2000 × 30% = 600 pixels wide.

Finally, the image must be positioned in the background so that it can slide behind the column to show only the amount of sidebar background necessary. This is done positioning it 70% from the left.

One True Layout

This thesis supports (and it is indeed one of its fundamental arguments) that a major liability of CSS for being used for layout is the lack of a true separation between presentation and content, and that this is mainly due to the dependence between the logical structure of the markup and the visual hierarchy of the document when it is rendered in a web browser. In other words, this thesis states that *the order of the content is dependent on the layout*.

There is an advanced technique, though, that could put this into question. Robinson (2005), in a seminal article, proved that it is possible to obtain a multi-column layout in any order regardless of their position in the document source code, using just standard CSS properties. Specifically, he addresses some of the same problems tackled by this thesis, to wit (Robinson, 2005):

Total layout flexibility

That is, the ability to order columns logically in the source while displaying them in any order desired. For any number of columns.

Equal height columns

Or more accurately, equal height columns without having to rely on faux columns.

Figure 12. The layout on which are based the examples of this section, where each `div` block represents a column.

```
<div id="content">
  <div class="fruit" id="orange">
    ...
  </div>
  <div class="fruit" id="strawberry">
    ...
  </div>
  <div class="fruit" id="lime">
    ...
  </div>
</div>
```

Vertical placement of elements across grids/columns

Designers face the choice of relying on elements being a particular height, resorting to tables or simply not bothering.

Although his technique, known as *One True Layout* allows any number of columns, for the following review I will focus only in a three-column layout (for more columns, the same principle would be applied). Thus, all the examples that follow are based on the markup shown in figure 12. After applying some styles for colours, fonts, etcetera, but without any layout property (neither float nor positioning), the result is shown in figure 13.

Of course, there are certain column orders that do not need any special treatment beyond the *normal*[1] use of floats for obtaining column layouts. Thus, a common 1-2-3 order would have been simply achieved with the following rules:

1 Note the italicised use of the adjective "normal". I am using it to refer to the nowadays common use of the `float` property to achieve this type of layouts. Nevertheless, as it was explained in the previous chapter, it was never intended for layout purposes.

Figure 13. Initial situation for the markup of figure 12, after applying basic styles of fonts, colours, borders, some paddings…, but with no layout properties (neither floats nor positioning).

Figure 14. By floating all the fruits to the left, constraining their width, and clearing the footer, it is very easy to obtaining a multi-column layout in the same order than the source document (1-2-3). However, the problem of equal-height columns is still unresolved.

```
.fruit {
  width: 33%;
  float: left;
}
#orange { width: 34%; }
#footer { clear: both; }
```

In this case I am using percentages for getting a liquid layout of three columns of equal width, but any other unit could have been used for the width of the columns. Note that the footer has been cleared, so that it is right placed below all the floated columns. However, as it can be seen in the figure above, the issue of getting equal-height columns is still unresolved. Because three liquid columns are being used, the aforementioned *faux columns* technique can not be applied

Figure 15. The same layout of figure figure 14, now with equal-height columns.

here. Robinson (2005) also addresses this issue in his article, using the following technique (the result is shown in figure 15):

```
#content {
  overflow: hidden;
}

.fruit {
  padding-bottom: 9999px;
  margin-bottom: -9999px;
}
```

Similarly, there are other combinations that can also be done with minor changes. For instance, an 1-3-2 layout would be as follows:

```
.fruit { width: 33%; }

#orange, #lime {
  float: left;
}

#strawberry { float: right; }
```

Yet another variants would be feasible just playing with the `left` and `right` values of `float` property, like 3-2-1 or 2-3-1. But, what does it happen if we want, for example, a 3-1-2 layout? In this case, simply floating the third column to the left would not work, because of the rules that govern float behaviour.

Here is when the main achievement of this technique: it basic principle is combining floats with *negative margins*. As it was mentioned on *Chapter 5*), negative lengths and percentages are perfectly legal values for `margin` and margin-related properties (although, again, they were intended for minor adjustments, like outdented paragraphs, hanging punctuation, or side notes, and some special effects, like slightly overlapping, and not for this use).

Though the whole technique will not be explained here in detail, the key is the following:

1 All columns are floated to the left.
2 Each column is placed at its desired position using `margin-left`, if needed (this is not necessary if it is immediately following the precedent one in the source code) *starting from the first column on the source code*.
3 Whenever a column must go leftmost than the previous floated columns, this is done using negative margins.

Since it is difficult to understand how it really works just by reading the above *algorithm*, I will use a concrete example to explain it. Let us suppose that we want an 3-1-2 layout. First, the foremost block in the source code must be placed. To do this, it is floated to the left and, since it must leave space for the third column to be placed at its left, a left margin of 33% is applied. After that, the second block has to be located. Because it is just to the right of the first block, it

does not require any special treatment: simply floating it does the work. The third column serves to illustrate the tough part of this technique: the other two floated columns plus the margin do not leave room for the third one (in fact, in this case they are occupying all the available width of the container):

> 33% (first column's left margin) 34% (first column's width) + 33% (second column's width) = 100%

Therefore, the third column will be placed *below* them (see figure 16).

Now, the third column is being rendered below the other two columns. But, for calculation purposes, we could think of it as if it were placed in a imaginary position at the right of second block (outside the visible area of the viewport). It is time to bring to scene **negative margins**. All we need to do then is to set a negative left margin equal to the sum of the previous space (in this case, 100%). The resultant CSS code will be:

```
.fruit {
  width: 33%;
  float: left;
}
#orange {
  width: 34%;
  margin-left: 33%;
}
#footer { clear: both; }
#lime {
  margin-left: -100%;
}
```

And the result is shown in figure 17.

Figure 16. The margin left applied to the oranges (first fruit in the source document) and the width of the oranges and strawberries do not leave room for limes at the top of the content. In addition, the float rules force a floated element to be placed as high as possible.

fruta rica

Lime is a term referring to a number of different fruits, both species and hybrids, citruses, which have their origin in the Himalayan region of India and which are typically round, green to yellow in color, 3–6 cm in diameter, and containing sour and acidic pulp. Limes are often used to accent the flavours of foods and beverages. They are usually smaller than lemons, and a source of vitamin C. Limes are grown all year round and are usually sweeter than lemons.

Limes are a small citrus fruit, Citrus aurantifolia, whose skin and flesh are green in colour and which have an oval or round shape with a diameter between one to two inches. Limes can either be sour or sweet, with the latter not readily available in the United States. Sour limes possess a greater sugar and citric acid content than lemons and feature an acidic and tart taste, while sweet limes lack citric acid content and are sweet in flavour.

The Valencia Orange is an orange first created by the Californian agronomist William Wolfskill, on his farm in Santa Ana. Its name comes from the Spanish city of Valencia, widely known for its excellent orange trees. The orange was later sold to the Irvine Company, who would dedicate nearly half of their land to its cultivation. The success of this crop in Southern California likely led to the naming of Orange County. The Irvine Company's Valencia operation later split from the company and became Sunkist. Cultivation of the Valencia in Orange County had all but ceased by the mid-1990s due to rising property costs, which drove most of what remained of the Southern California juice orange industry into Florida.

Primarily grown for processing and juice production, Valencia oranges have seeds, varying in number from zero to six per fruit. However, its excellent taste and internal color make it desirable for the fresh markets, too. The fruit has an average diameter of 2.7 to 3 inches (70 – 76 mm). After bloom, it usually carries two crops on the tree, the old and the new. The commercial harvest season in Florida runs from March to June. Worldwide, Valencia oranges are prized as the only variety of orange in season during summer.

Fragaria (pronounced /fra'geəriə/) is a genus of flowering plants in the rose family, Rosaceae, commonly known as strawberries for their edible fruits. Originally straw was used as a mulch in cultivating the plants, which may have led to its name. There are more than 20 described species and many hybrids and cultivars. The most common strawberries grown commercially are cultivars of the Garden strawberry (*Fragaria ×ananassa*). Strawberries have a taste that varies by cultivar, and ranges from quite sweet to rather tart. Strawberries are an important commercial fruit crop, widely grown in all temperate regions of the world.

Figure 17. The 3-1-2 final layout.

7

Case Studies

Previous chapters have focused on reviewing current CSS mechanisms , as well as more advanced techniques that push CSS2 to its limits, in an attempt to achieve a greater independence between the structure of the document and its visual layout.

This chapter presents some case studies, to demonstrate that such separation is not yet possible, and that more advanced layout mechanisms are needed to meet the designers' requirements.

Mismatch between Content Order and Visual Position

It is not necessary to turn to elaborated examples to reveal some of the problems of CSS that prevent it from being considered a true layout language. One of such issues is the *dependence between the document source order and its layout*, or, put another way, *between the document structure and the visual structure*. This can be shown with a very simple scenario, like a blog post or a news article in a newspaper.

Blog Posts

Blogs are a good tool to illustrate how current CSS layout mechanisms are far from providing a true separation between presentation and content. Whereas most blogs share a common structure, their markup differs from one to another and tend to imitate, to a greater or lesser extent, the same order than their layout. Focusing only on blog entries, and although the concrete elements may vary, they contain essentially the same information in every blog, namely:

- Title of the post
- Date of publication
- Categories under which it is published
- Tags
- Number of comments to that entry
- The content itself

However, the markup is often very different from one blog to another, specially with regards to the order in which the above elements appear on the document source code. Just as an example, let us consider the following blogs, which have been chosen on purpose belonging to renowned web designers, very compromised with web standards and also deeply-versed in CSS:

- zeldman.com (Jeffrey Zeldman; see figure 1)
- stuffandnonsense.co.uk (Andy Clarke; see figure 2)
- meyerweb.com (Eric Meyer; see figure 3)
- www.markboulton.co.uk (Mark Boulton; see figure 4)

- stopdesign.com (Douglas Bowman; see figure 5)
- jasonsantamaria.com (Jason Santa Maria; see figure 6)

For illustrative purposes, I will focus only on the title of each post and the date when they have been published. If we think on the markup to represent a blog entry solely from a structural point of view, leaving aside presentational considerations, most of us would agree that the right order for such elements would be: first the title of the post and then the date of publication (probably together with the other metadata of the entry such as the categories to which it belongs, the number of comments, etcetera). Therefore, a possible markup for blog entries would be as follows:

```
<div class="entry">
  <h1>Lorem Ipsum</h1> <!-- Title of the post -->
  <span class="date">11 <abbr
title="December">Dec</abbr> 2009</span>
  <div class="content">
    ...
  </div>
</div>
```

As it can be seen, the blog entry is enclosed in a `div` block, thus grouping all the elements that form the blog entry, because they are a related content that can be seen as a logical section of the document. This block is given a significative class name: `entry` (other names like `post` or `article` would have been also appropriate). The title of the post is represented with a `h1` element, and it is followed by the date of the entry. However, many blogs follows the newspaper tradition of placing the date in front of the title, as in the blogs of Zeldman (figure 1), Boulton (figure 4), and Santa Maria (figure 6). Although this seems to be a minor variation with respect the suggested markup above, the truth is that the source code of each of the reviewed blog posts is tightly coupled to their layout.

Thus, Zeldman puts the date before the title of the post:

Figure 1. A blog entry by Jeffrey Zeldman.

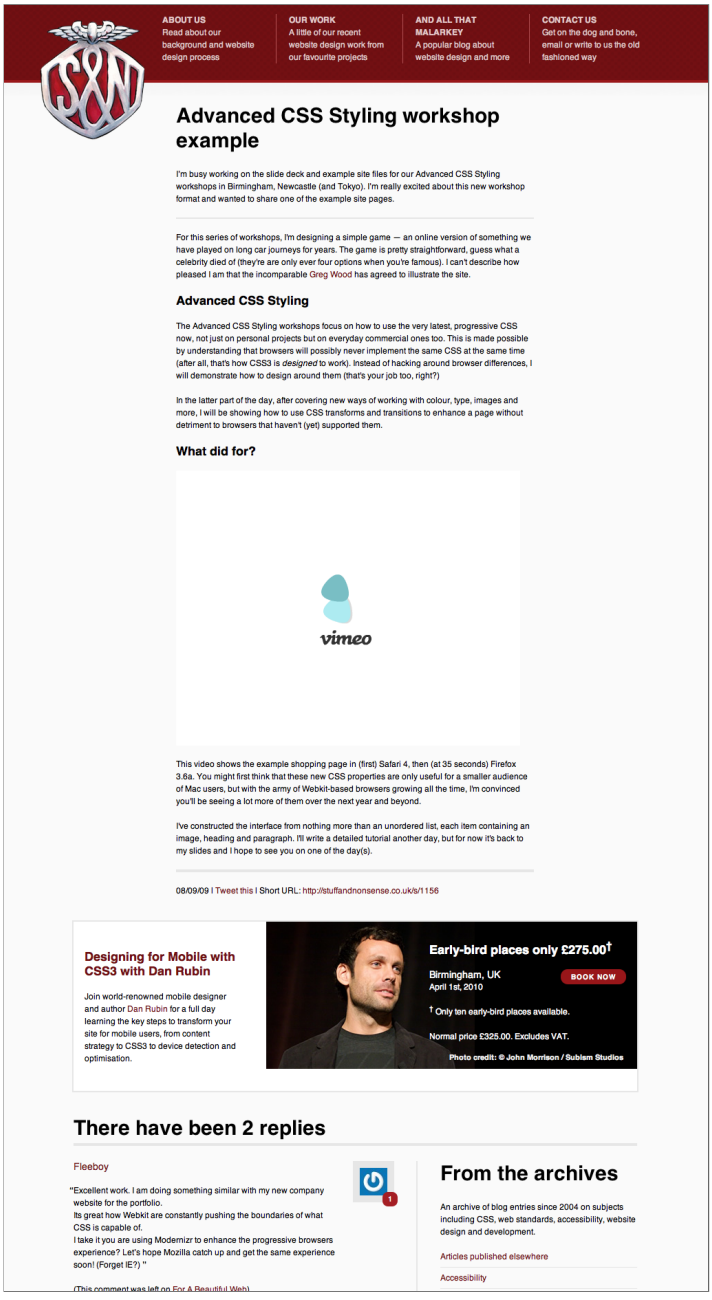
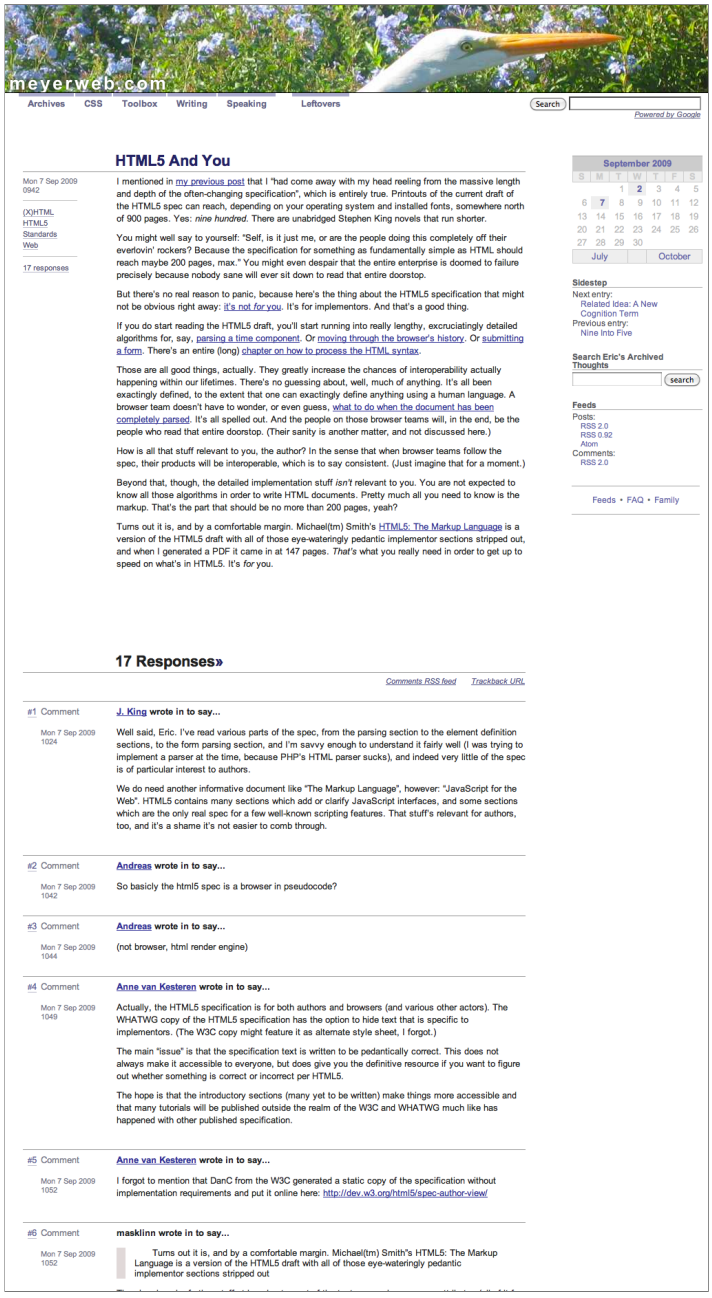
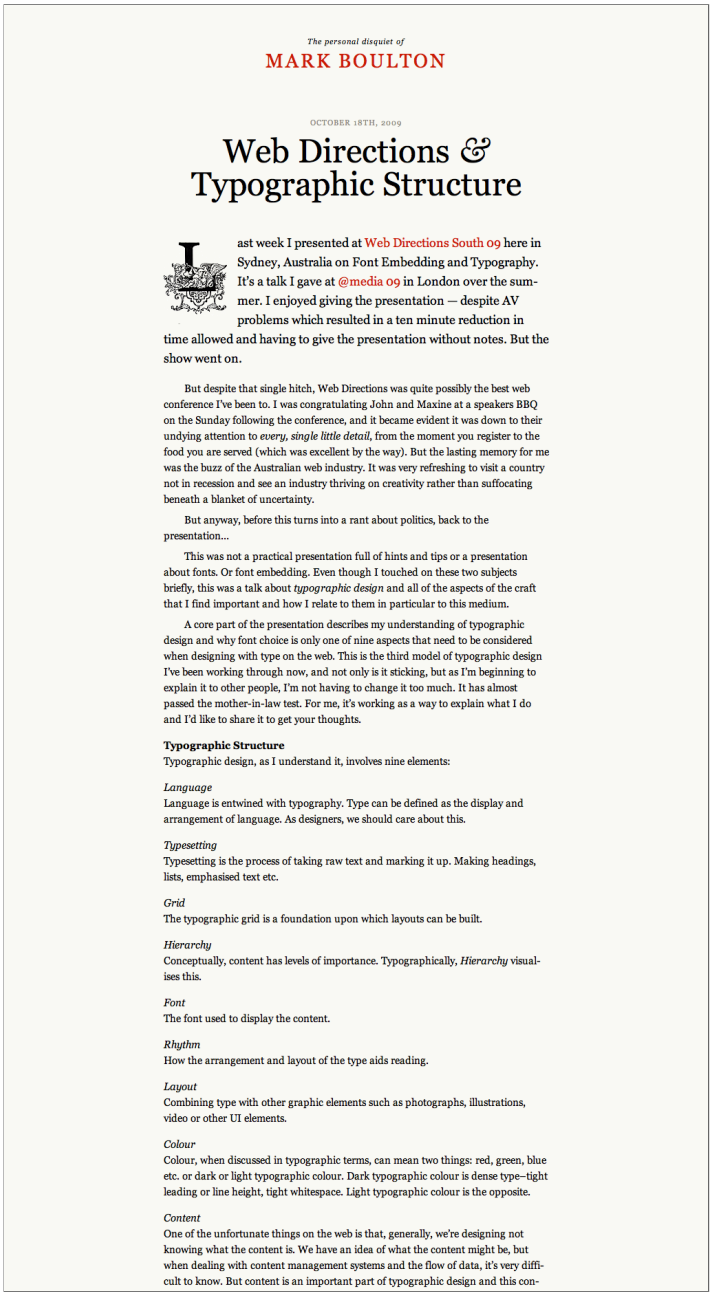


Figure 2. A blog entry by Andy Clarke.

Figure 3. A blog entry by by Meyer.





The personal disquiet of
MARK BOULTON

OCTOBER 18TH, 2009

Web Directions & Typographic Structure

Last week I presented at Web Directions South 09 here in Sydney, Australia on Font Embedding and Typography. It's a talk I gave at @media 09 in London over the summer. I enjoyed giving the presentation — despite AV problems which resulted in a ten minute reduction in time allowed and having to give the presentation without notes. But the show went on.

But despite that single hitch, Web Directions was quite possibly the best web conference I've been to. I was congratulating John and Maxine at a speakers BBQ on the Sunday following the conference, and it became evident it was down to their undying attention to *every, single little detail*, from the moment you register to the food you are served (which was excellent by the way). But the lasting memory for me was the buzz of the Australian web industry. It was very refreshing to visit a country not in recession and see an industry thriving on creativity rather than suffocating beneath a blanket of uncertainty.

But anyway, before this turns into a rant about politics, back to the presentation...

This was not a practical presentation full of hints and tips or a presentation about fonts. Or font embedding. Even though I touched on these two subjects briefly, this was a talk about *typographic design* and all of the aspects of the craft that I find important and how I relate to them in particular to this medium.

A core part of the presentation describes my understanding of typographic design and why font choice is only one of nine aspects that need to be considered when designing with type on the web. This is the third model of typographic design I've been working through now, and not only is it sticking, but as I'm beginning to explain it to other people, I'm not having to change it too much. It has almost passed the mother-in-law test. For me, it's working as a way to explain what I do and I'd like to share it to get your thoughts.

Typographic Structure
Typographic design, as I understand it, involves nine elements:

Language
Language is entwined with typography. Type can be defined as the display and arrangement of language. As designers, we should care about this.

Typesetting
Typesetting is the process of taking raw text and marking it up. Making headings, lists, emphasised text etc.

Grid
The typographic grid is a foundation upon which layouts can be built.

Hierarchy
Conceptually, content has levels of importance. Typographically, *Hierarchy* visualises this.

Font
The font used to display the content.

Rhythm
How the arrangement and layout of the type aids reading.

Layout
Combining type with other graphic elements such as photographs, illustrations, video or other UI elements.

Colour
Colour, when discussed in typographic terms, can mean two things: red, green, blue etc. or dark or light typographic colour. Dark typographic colour is dense type–tight leading or line height, tight whitespace. Light typographic colour is the opposite.

Content
One of the unfortunate things on the web is that, generally, we're designing not knowing what the content is. We have an idea of what the content might be, but when dealing with content management systems and the flow of data, it's very difficult to know. But content is an important part of typographic design and this con-

Figure 4. A blog entry by Mark Boulton.

Figure 5. A blog entry by Douglas Bowman.

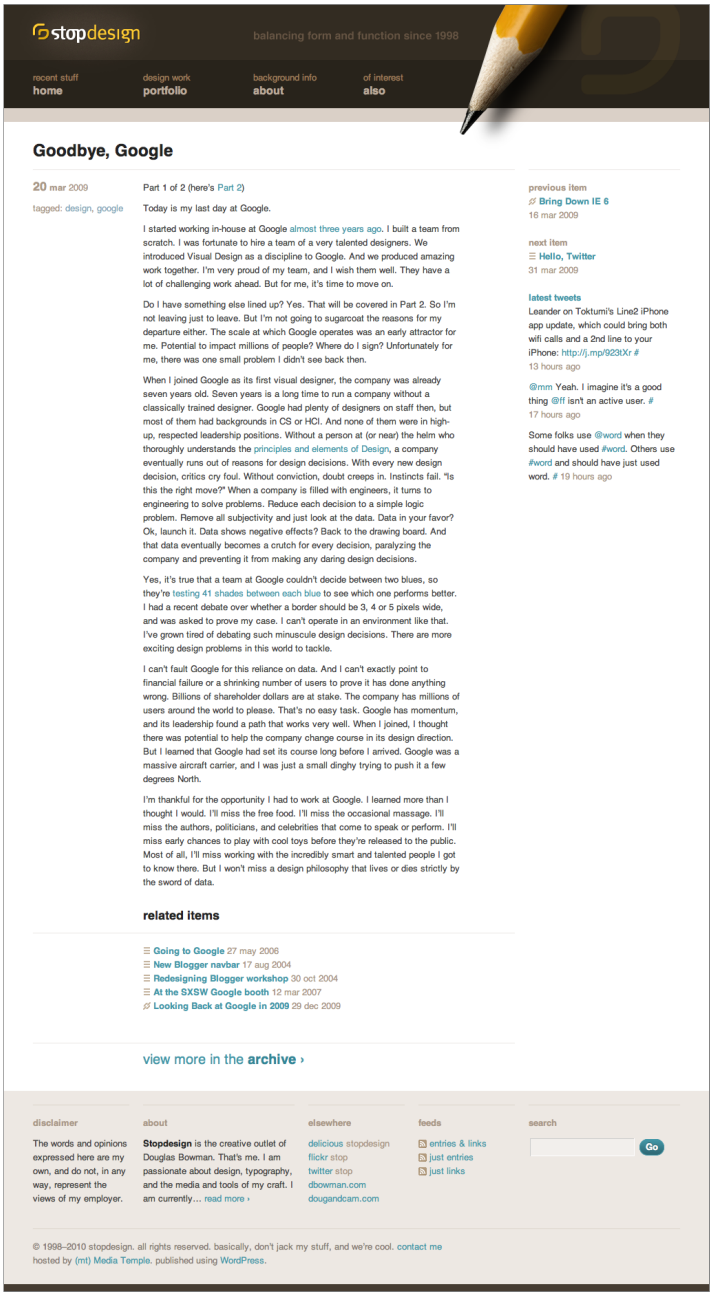


Figure 6. A blog entry Jason by Santa Maria.

```
<div class="timestamp">
  <h3><a href="http://www.zeldman.com/2010/01/08/
aigany-members-series-the-one-that-got-away/">8 Jan
2010</a> 8 am eastern</h3>
</div>
<h2>AIGANY / MEMBERS SERIES: THE ONE THAT GOT AWAY</h2>
```

There is no doubt, though, that Zeldman agrees with the logical structure suggested above, since he is using an `h2` heading for the title and a `h3` heading for the date. That is, his code is showing that he actually considers the date to be logically contained into the blog entry. However, the order or the markup follows that of the visual layout

The same happens in Boulton's blog, other author that also places the date above the title of the post:

```
<div class="post" id="post-812">
  <div class="posttitle">
    <p class="date">October 18th, 2009</p>
    <h2>Web Directions <span class="amp">&amp;amp;</span>
Typographic Structure</h2>
  </div>
  ...
</div>
```

The last of the reviewed blogs that set the date above the title, Santa Maria's one, is however using a similar HTML structure to what I proposed at the beginning of this section, where the date (and in this case other metadata) follows the title:

```
<h1 class="post-head">In Sugar We Trust</h1>
<div id="post-meta-top">
  <ul>
    <li class="date"><strong>2009</strong> <span>Oct
26</span></li>
    <li class="comms"><a
href="#comments"><strong>Comments</strong>
<span><em>38</em></span></a></li>
    <li class="cats"><strong>Published In</strong> ...
/li>
    <li class="prev"> ... </li>
    <li class="next"> ... </li>
  </ul>
</div>
<div id="content"> ... </div>
```

So it seems that CSS does allow, after all, this type of changes in the position of the elements on the page without compromising the logical order of the content. Unfortunately, this is not usually a *bullet-proof* solution.

To understand how Santa Maria has achieved this (and its weaknesses), I am going to do the same for the more reduced HTML code of my example that was shown on page 133.

This type of changes between the structure of the markup and the arrangements of the elements on the screen is only possible by means of **absolute positioning**. The first step is therefore to take the date out of the normal document flow and position it before its preceding `h1` element in the source code:

```
.entry { position: relative; }
.date {
  position: absolute;
  top: 0;
}
```

Note that the first rule is needed to make the containing `div` for the entry to become the containing block for the absolute posi-

Figure 7. When the browser text size is increased, absolute positioned elements may overlap the rest of the elements of the page (either positioned or not), or be hidden by them, depending on the order they have been defined in the document source order. In this case, Santa Maria is taking a list with the blog entry metadata out of the normal flow and putting it above the title of the post. When the browser text size is increased enough, content overlaps.

tioned element (otherwise, it would be positioned at the top of the the page, or the first positioned ancestor element). Now, there is needed to make room for the date not to overlap the entry title:

```
.entry { padding-top: 1.6em; }
```

Actually, the value of the top padding is just an example: it will depend on the font size of both title and date, the wanted space between them and the containing div, whether the date fits in one line or text or can expand more, etcetera. The problem is that, as long as absolute positioning is involved, the solution does not work always, under ever circumstance. Returning to the Santa Maria example, if we increase the font size of the browser, the absolute positioned list with the metadata of the entry overlaps the content below it, as it is shown in figure 7.

Newspaper Headlines

The same that have been said here for the titles of blog posts and their publication date also applies to the headline and summary of a news article in a newspaper. Although the summary usually follows the headline, sometimes, designers of the newspaper decide to put it above the headline, as in the six-column headline of the Canadian newspaper that is shown in figure 8 (Berry, 2004, p. 46). In the main news article of that page, the headline comes ofter the summary. Nevertheless, it is still clear what the headline is, thanks to its much bigger size and the use of a bold face, and there is little doubt that reader's eyes will be attracted first to the headline, and only then he will probably focus on the summary. In this case, there is no doubt

A12 WORLD NATIONAL POST, MONDAY, MAY 17, 1999

OFF THE TOP

Gunman on the run after killing five • French and German authorities launched a cross-border manhunt yesterday for a gunman on the run after killing five people in a pre-dawn rampage that began outside a disco in Germany, French police said. The suspect is known to police and was "heavily armed and would not hesitate to use his weapons," said a German police spokesman. The gunman shot two people outside the bar and another two in an apartment building in Dillingen, before entering France and killing a fifth person in Sierck-les-Bains, French police said.

Guatemalans vote on reforms • Guatemalans went to the polls yesterday for the first time since the end of the country's bloody civil war in 1996, in a national referendum on constitutional reforms. If approved, the 50 constitutional reforms would recognize the rights of the Mayan Indian majority for the first time since Europeans arrived in the 16th century.

Four charged with planning school killing spree • Four boys have been charged with plotting a shooting at their Port Huron, Mich., school similar to the rampage at Colorado's Columbine High School. Justin Schoepp and Jedaiah Zinno, both 14, were charged as adults Thursday Schoepp and Jedaiah Zinno, both 14, were charged as adults Thursday with conspiracy to commit murder for allegedly planning to kill classmates at their 760-student Holland Woods Middle School. The two others, ages 12 and 13, were charged as juveniles. Their plans were discovered after a girl overheard them talking about a killing spree.

Since 1913, Thomas Jefferson's descendants have gathered at their ancestral home in Virginia. This year, the family bad boy invited 35 progeny of the president's slave mistress, welcoming them as 'members of our family'

A tangled tale of kinship and race

By Jan Cienski

CHARLOTTESVILLE, VA. - Thomas Jefferson — author of the Declaration of Independence, the third U.S. president, noted scholar, and the owner of more than 200 slaves — has become a symbol of America's centuries-old fixation on race.

He had a white wife, Martha, with whom he fathered two daughters, and a mulatto slave mistress, Sally Hemings. Even during Jefferson's life, rumours circulated that he had fathered several children with his slave.

Most historians dismissed the claims and Jefferson's official descendants scoffed at the notion that the father of their country could also have been the father of illegitimate slave children.

But Hemings' descendants insisted that Jefferson was their ancestor. Last year, DNA tests indicated that at least one of her seven children had been fathered by a Jefferson — although it's still not clear if that male is Thomas or another relative.

Over the weekend at Monticello, Jefferson's Italianate hilltop estate in the rolling green countryside of central Virginia, the descendants of both Hemings and Jefferson finally met and tried to figure out if they are family.

Every year since 1913, Jefferson's acknowledged descendants have gathered at their ancestral home. The family's Monticello Association administers the Jefferson family graveyard located on the estate. All 200 members — and hundreds more acknowledged Jefferson descendants who don't belong to the group — have the right to be buried alongside their forefather and 100 other relatives in the sun-dappled cemetery.

The mostly black descendants of Sally Hemings are not members of the association and have no right to be buried in the graveyard that their slave ancestors helped build.

But this year for the first time, a family rebel invited 35 Hemings' descendants to the Monticello reunion.

"I welcome them to this place and on to this land as members of our family," Lucian Truscott IV, a Los Angeles author and screenwriter, said Saturday during the opening ceremony at Monticello.

As he spoke, dozens of well-dressed people, both black and white, strolled the lush lawns of Monticello sipping glasses of wine.

Mr. Truscott, a sixth-generation grandson of Thomas Jefferson whose mother is buried in the graveyard, wants the Monticello Association to extend membership to the Hemings' branch, granting them the same rights and privileges that the white family enjoys.

Michelle Cooley-Quille says it's about time that her family is accepted by the larger clan. Ever since she was a small girl, she remembers hearing family tales of her ancestry.

"When I first told my friends at school, I could see their surprise at how someone who is black could be related to someone who is white," said Ms. Cooley-Quille, 33, a clinical psychologist at the Johns Hopkins School of Public Health in Baltimore.

Her father, Robert Cooley III, a Vietnam veteran and a judge, died last summer and his last wish was to be buried at the plantation. He was turned down because the association could not trace him definitively to Jefferson.

Now his daughter insists she wants to be buried at Monticello.

"Half my ancestors were enslaved here and the other half of half my ancestors created this place," she said.

But there is no rushing tradition and genealogy. The association plans to put the matter of accepting Hemings' descendants to a vote of all its members over the next year. The concern is that the Hemings' oral histories and DNA tests are inconclusive, said Robert Gillespie, the association's president.

"We are looking at what criteria could be adopted that would bring them into the association," said Mr. Gillespie, an attorney who works in the state capital, Richmond. "Heretofore we used the genealogy from Jefferson's two daughters. That genealogy doesn't pick up any descendants from Sally Hemings."

Jefferson's affair with his slave has caught the popular imagination in America. After being a taboo subject for many years, it has become widely accepted that he had an affair with Hemings.

"The general public probably does think that the affair happened, but from the family's perspective we had our own oral traditions that the [Hemings] children were fathered by his nephews," said Mr. Gillespie.

As early as 1802, a political rival published a newspaper account of a liason between the president and his slave — a scandalous accusation in those years.

Jefferson never made any public response to the charges and Sally Hemings left no known account of the matter. But at least two of her children claimed Jefferson was their father.

Historians slowly began chipping away at the story and last year, a DNA test found that a descendant of Eston Hemings, Sally's youngest son, had the same Y-chromosome as the male line of the Jefferson family. Although the test showed a relationship, it could not pinpoint exactly which Jefferson had fathered Eston.

A similar test on the descendants of Thomas Woodson failed to find a genetic link, although a new blood sample is now being tested. The descendants of Hemings' other children can't be tested because DNA tests for paternity can only be determined on the male line.

The story is that the same genetic tests couldn't prove paternity for the blue-blooded acknowledged family because they are descended from Jefferson's two daughters — Martha Randolph and Maria Eppes. What the acknowledged family does have is birth certificates showing parentage, which weren't issued to slaves.

That leaves any question of admitting the descendants of a slave up in the air, said David Works, 42, a past membership committee chairman and a descendant of Martha Randolph.

"A descendant has to be declared by the father or by blood," he said. "I can't make them an heir."

That's a double standard, said Ms. Cooley-Quille. "I am willing to provide my blood when the Monticello Association provides theirs," she said.

For Mr. Truscott, the family bad boy, all this talk of proof is simply a way for the association to avoid acknowledging that they have black cousins and that the great man used his slaves for sex.

"In this family, the word genealogy has become a code word for race," he said angrily, insisting that he will continue to personally invite the Hemings' descendants every year — even if the association fails to grant them membership.

Mr Truscott says it is simple justice to welcome the descendants of the slaves who fired every brick, forged every nail, and cut every board used to build Monticello.

"On the morning that Mr. Jefferson wrote the Declaration of Independence, someone woke him up, someone got him dressed, someone fried him an egg, someone prepared his pen. They were slaves," he said. Without those slaves, Jefferson would never have found the time to write, "We hold these truths to be self-evident: that all men are created equal ..."

Just down the slope from the graceful, neoclassical Monticello lie the outlines of the dirt-floored huts that had housed the plantation's slaves.

The tangled tale of kinship, blood, and race shows that America's racial roots are a lot more complex than many acknowledge.

Those DNA tests linking the Hemings' descendants to the Jeffersons were done on a family from Staten Island, N.Y. The family is white.

Eston Hemings was one of only a handful of slaves freed by Jefferson in his will. He eventually made his way to Wisconsin, changed his name to Jefferson and, taking advantage of his light skin, slipped across the colour line to live as a white man.

His mother, Sally Hemings, was very light-coloured. That's because she was the slave half-sister of Jefferson's beloved wife, Martha. Her father, like many other slave owners, had unacknowledged children that were kept in bondage.

Martha died after only 10 years of marriage and it is said that a distraught Jefferson turned to Hemings, who resembled her free, white sister.

That kind of racial fuzziness makes a lot of white Americans uncomfortable, said Russell Adams, chairman of the Department of Afro-American Studies at Howard University in Washington.

"The larger society represented by the Jefferson family association does not wish to admit biological and social relationship to the black community, whose ancestral members were property along with horses, cows, and pigs," he said.

At the family reunion, a young, black Hemings' descendant, with a baseball cap perched sideways on his head, called for all members of the family, both black and white, to gather on the steps of Monticello for a photo. About 80 people crowded together but dozens more stayed on the lawn, basking on.

"I think it's inappropriate," to jump to the conclusion everyone is related, said Mr. Works, as he watched the crowd on the stairs. "If it doesn't work out, you're going to have a lot of people with hurt feelings."

National Post

Descendants of Thomas Jefferson and descendants of his slave Sally Hemings pose for a group shot at Jefferson's plantation during the Monticello Association's annual meeting in Charlottesville, Va., on Saturday.

Lucian Truscott IV a sixth-generation white grandson of Thomas Jefferson, says the Jefferson family association should admit the black descendants of Sally Hemings, one of Jefferson's slaves.

U.S. Army recognizes witchcraft as religion

WICCANS GET CHAPLAINS

By Ben Fenton

WASHINGTON - The United States Army has recognized white witchcraft as a religion and has appointed chaplains to oversee pagan ceremonies at least five bases.

A Pentagon spokesman said Thursday there were believed to be at least 100 witches attending covens at Fort Hood, Tex., the army's largest base with more than 42,000 troops.

So respectful has the army become of the pagan rites that security was increased at Fort Hood's Boy Scout camp, where covens meet.

The move is to deter members of Christian groups from intimidating the group.

The pagans, called Wiccans, are accorded the same privileges as practitioners of Christianity, Judaism, and Islam.

They are encouraged to have their religious preference stamped on the metal dog tags each soldier wears.

Lieutenant-Colonel Donald Treyer, the Seventh Day Adventist army chaplain who has been given responsibility for Fort Hood's coven, admitted he was not overjoyed with his job because fellow Christian pastors disapproved and had been "cool" toward him.

"It's such a volatile subject," he said. "It just sparks a fury."

The Pentagon says it has received several requests for a posting to Fort Hood because it has such a large pagan congregation.

"We are obliged by the Constitution to respect and make provisions for the religious needs of members of the military and not to pass judgments on their beliefs," a spokesman said.

The Covenant of the Goddess, which claims to represent most American Wiccans from its base in Berkeley, Calif., says there are about 50,000 followers of the faith in the United States.

They celebrate earth-spirits such as the "great goddess Freya" and on their altars give blessings to water, bread, and salt, the three essentials of life, while the congregation holds hands in a circle around a large bonfire.

Their main festivals are at the vernal and autumnal equinox and at midsummer.

However, sacrifices, either human or animal, are not tolerated. "It is not something we do," said one Wiccan.

The Daily Telegraph

American museum exposes phoney glamour of drugs

Exhibits tell story of U.S.s century-long battle for and against narcotics

By Hugo Gurdon

PENTAGON CITY, MD. - The most telling exhibits in the U.S.'s newest museum are burnt and bent teaspoons, stained rags, used soda bottles, and a diamond-encrusted Colt 45.

They expose the sordid, deadly reality and phoney glamour of their subject, which is drugs.

Gathered in glass display cases are the paraphernalia of the U.S.'s century-long battle for and against the right to "get high" — bongs (water pipes), psychedelic posters, hornets rolling papers, Tommy guns and grenades.

The Drug Enforcement Agency museum, which opened its doors to the public last week, lays bare the willful self-delusion of the 1950s and 1970s, when baby boomers swept aside a mass of historical evidence and argued drugs were intrinsic to life, liberty, and the pursuit of happiness, rather than the low road to ruin.

One of the exhibits is a 1937 poster, captioned "The Devil's Roost," in which the vultures of "vice," "want," "insanity," "misery" and "crime" perch on a reaper-like figure marked "Dope." A few feet away is a 1908 *New York Times* front page taken up with a story under the headline "The growing menace of the use of cocaine."

Dominating one wall is a black and white picture of the actress Dorothy Davenport, "shooting up" with a syringe in the 1923 film *Human Wreckage*. More than 100 feature films were made in the silent era focusing on the perils of drugs.

In 1900, one in every 200 Americans was a drug addict. Many of them swigged laudanum.

By the Second World War, though, drug addiction was a marginal, statistically-negligible vice in the U.S. Decades of evidence, and policing by federal agents armed with a Tommy gun and two grenades each as standard issue, had worked.

Then came the 1960s and gurus such as Allen Ginsberg, demonstrating in New York with a poster bearing the legend "Pot is a reality kick." The drug culture was born again.

Traditional corner stores, which sold everything from ice cream to toothpaste, gave way to "head shops" that peddled drug paraphernalia, such as hookahs and roach clips (tweezers to hold a marijuana cigarette when it was too small and hot for bare fingers).

Redolent of the drug culture's seediness are the opium-stained rags, pipes made from old 7-Up bottles, and elasticated armbands to make a junkie's veins bulge to make injections easier. So, too, are the photographs of dealers shot dead, and addicts in the final contortions of a fatal overdose.

Perhaps the most telling artifact in the museum is the gleaming, silver Colt 45 handgun, with the diamond-studded grip.

It was owned by Rafael Cerro Quintero, a Mexican drug baron. It is the quintessence of the phoney heroism and deceptive glitter at the heart of the drug culture.

Mr. Cerro Quintero glares out from a photograph nearby, dishevelled and dressed in nothing but dirty jeans at the time of his arrest. He murdered a DEA agent and is now behind bars.

The Daily Telegraph

Among the drug paraphernalia used in 1970 is this homemade bong on display at the Drug Enforcement Administration's new museum. The bong, a water pipe for smoking marijuana, was seized by agents in the field.

Figure 8. A page from canadian newspaper *National Post*. In the main news article, the summary is before the headline, whereas in others it is below. However, it is clear that if it that page had been written in HTML, the structure of a news article should remain the same. Picture from the *Contemporary Newspaper Design* (Berry, 2004, p. 46)

that (if we concentrate only in that article, ignoring the rest of the elements of the page), the order of the equivalent HTML markup would have been[1]:

```
<h2>A tangled tale of kinship and race</h2>
<p class="summary">Since 1913, Thomas Jefferson's
descendants ... </p>
<p class="author">By <cite>...</cite></p>
...
```

However, in other news articles, the summary is below the title. And this happens even on the same page of the same newspaper. That is the case of the article titled "American museum exposes phoney glamour of drugs". Why should both news be represented differently? And yet, as for the case of blog posts, even these minor variations in the layout over the structure of the underlying HTML are only possible in CSS through absolute positioning, which, as have already been stated, does not guarantee that it works for every situation. On the contrary, usually even a simple change on the font size may cause overlapping issues; a liquid layout could not be used; and —which is particularly problematic for the case of newspapers, usually based on content management systems (CMS)— it prohibits the use of *templates*, because the text of the summaries will probably be very different from some news to others and occupy different number of text lines, thus incurring overlapping, unless manual adjustments be made for each specific case. But… where is the separation between presentation and content then?

1 Note that, as for the previous blog post examples, I am focusing only on the order of the content, here. Although the structure is also important, of course, the concrete elements chosen for representing such structure may vary from one author to another. For example, I am using a normal paragraph with a "semantic" name for the summary, because in my opinion it is more right than representing it with a subheading (an `h3`, in this case). But other authors could argue the opposite. Similarly, the level of the heading that represents the headline is usually not a white and black issue. Anyway, the concrete elements are, to a great extent, irrelevant for what I am trying to shown here: the dependency between the layout and the order and structure of the content (what elements are children of others, which ones follow which others, etcetera).

Floating a Block

The title of this section, which may be too abstract, aims to reflect that CSS `float` property lacks the power of the similar features that can be found in desktop publishing tools, even if we consider as such applications more oriented towards the domestic user and word processing, such as Microsoft Word or Apple Pages. Whereas in those tools the user can specify many options for how the text must flow around a floated object, in CSS we are limited to float it to the left or to the right. This, along with the fact that floats in CSS are highly dependent on the order and structure of the source code, limit the type of layouts that we can currently do with Cascading Style Sheets.

To illustrate the stated above, I will make use of two examples. The first one is a website that I did a few years ago for a new master in web engineering that we started to teach at the Computer Science School of University Oviedo in 2006. The second example comes from a local magazine of the region of Biot (France). When I was starting to write this dissertation, during my short research stay at W3C Office in Sophia Antipolis, one day, after lunch, while we were drinking a coffee sitting on the sofas in the pleasant living room of our office, I picked up the Summer 2009 issue of BIOTinfo magazine and —I already had begun to see layouts *everywhere*— said: "Look, Bert, a very simple layout that however can not be made in CSS."

"L"-shape Layouts

The first example is shown in figure 9. It is a small, static web site that I did for promoting our master in web engineering. It uses a three-column hybrid layout, where the main content is in the middle column, and left and right columns contain highlighted information about dates, contact information, whom is the master oriented, etcetera. Sidebars have an fixed width (defined in pixels), while the main content is liquid (it expands to fit the available width). Finally, the maximum available width is limited in the container using a `max-width` property.

Figure 9. Website of the Master in Web Engineering at University of Oviedo (as it was seen in 2006, when I designed it). It uses a three-column layout, and my initial purpose was that the middle column, where it is the main content, expanded under the left one to maximise the available width when the page was rendered on a small browser window, making a sort of "L" shape. But this is not currently possible with floats in CSS (without changing the logical order of the source code).

My initial purpose was that main content expanded beneath the highlighted boxes of the left column, thus making a sort of "L"-shape. Leaving aside aesthetic considerations[1], this would make sense for two reasons:

- The main content is the area which content is more probable to grow, while the other two columns should usually remain more or less the same.

1 Unfortunately, I am not a graphic designer —despite how much I would like to be— and therefore neither this example nor any other of my own examples in this dissertation pretend to be an example of good design, and they must only be taken for illustrative purposes.

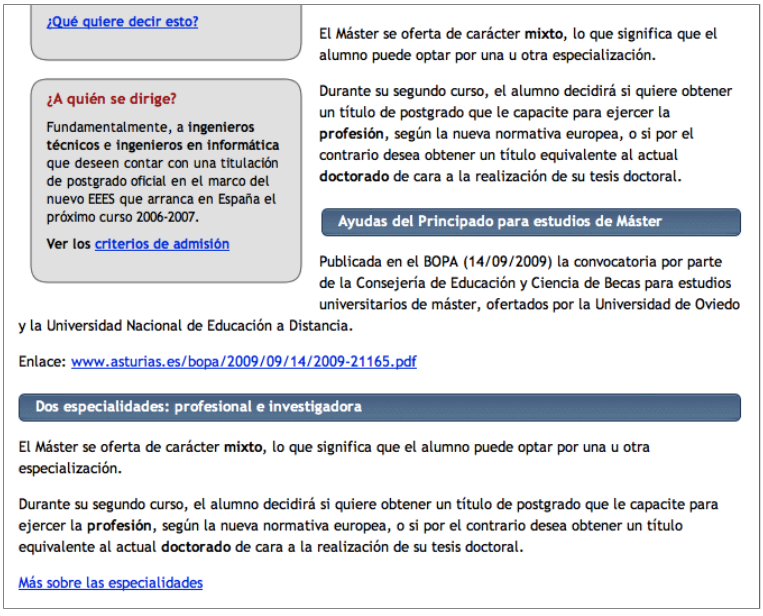


Figure 10. This is the initial desired behaviour for the Master in Web Engineering (MIW) website: the liquid main content area should expand below the highlighted content to fit all the available width of the page container, thus improving the user experience in small screens.

- Until a mobile version were made, I wanted that the website were reasonably legible in a small screen, or with a reduced browser window.

My first assumption when designing the information architecture of the site has proven to be truth. Although I do not maintain the website anymore, its design has been kept, and the content in the home page has increased a lot, as can be seen in its current online version[1].

As for the second requisite, I would have liked that the layout behaved as it is depicted in figure 10.

The only tool that allows this type of layouts in CSS is float. In fact, this is quite similar to the intended purpose of floats: the only difference is that I am floating some pieces of content, enclosed in `div` blocks, instead of an image. But apart from that the scenario is the same.

There is, though, a problem with float that prevent it to be used for achieving effects like that, and that has already been mentioned

1 http://www15.uniovi.es/master/ingenieriaweb/

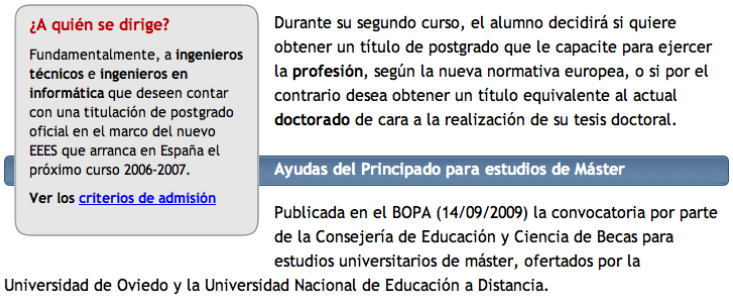


Figure 11. Backgrounds and borders expand beneath the floated elements. This is the right behaviour according to the specification, because float shortened the line boxes of the surrounding elements, not the boxes themselves. That is, is the content of an element, and not its generated box, that flows around a floated element.

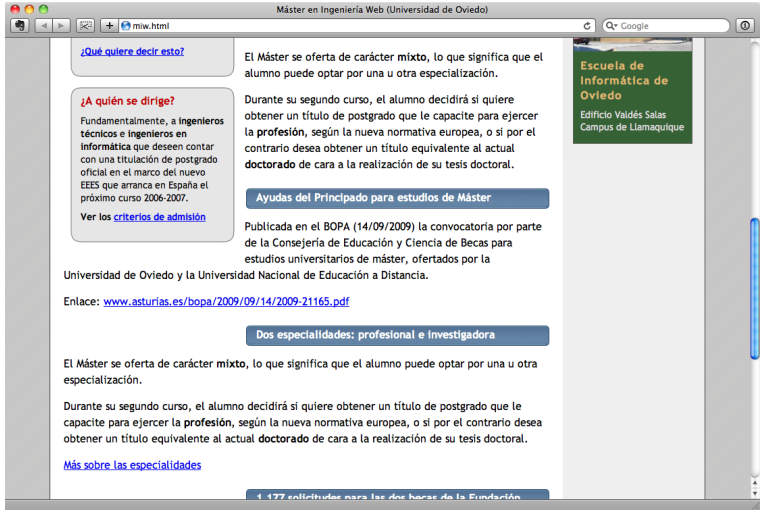


Figure 12. After applying a left margin to the headings of the main content area, we avoid that they are hidden beneath the rounded floated boxes, but we have gained another undesired effect.

on *a previous chapter*. It is what I titled *What Flows Is the Content, Not the Box Itself* (see p. 104). Due to what was described there, if we simply did something like this:

```
.highlighted {
  float: left;
  clear: left;
  width: 243px;
  margin: 0 0 15px;
}
```

Then we would obtain the result shown in figure 11. The only way of avoiding that is to specify a margin left for at least the rounded titles:

```
h3 {
  margin-left: 270px
}
```

But then we would not obtain the desired layout shown in figure 10, but the unpleasant one shown figure 12.

Another Example

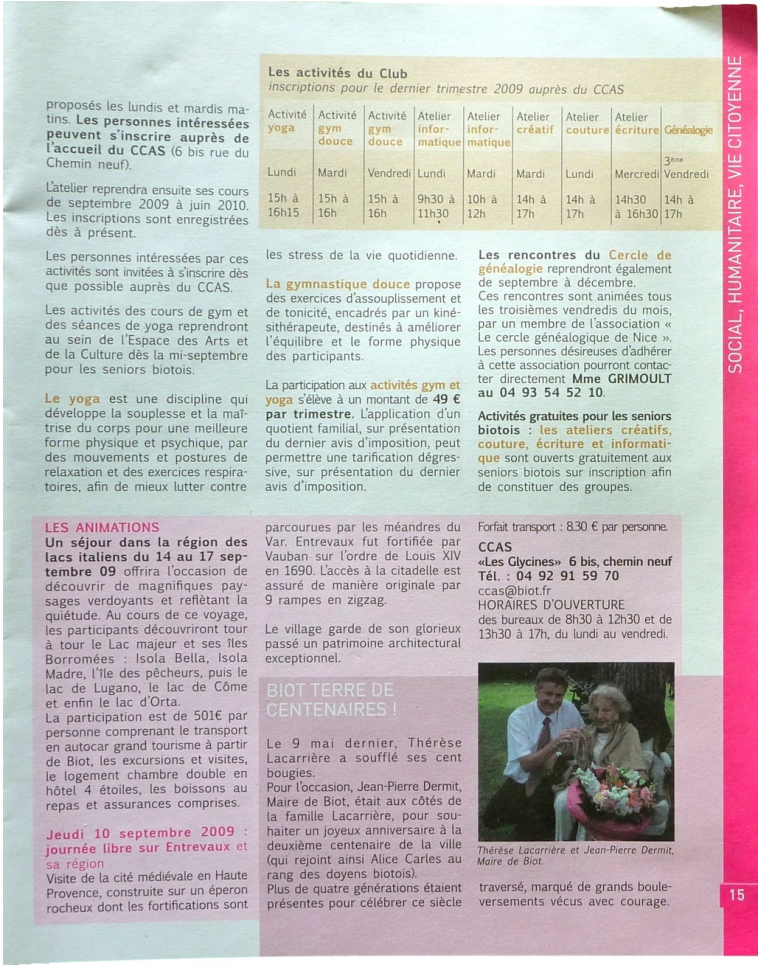
Another example of the inability of floats to carry out certain layouts can be found in figure 13. It is a page of BIOTinfos, the municipal magazine about the village of Biot (Sophia Antipolis, France). Specifically, I am going to focus on the two pink boxes that appears at the bottom of the page: *Les animations* and *Biot terre de centenaires!*.

Once again, there is probably not a single right markup for that content. Anyway, this is to a large extent irrelevant for this example: once chosen one markup, if separation between presentation and content were a reality, the final layout should be doable with CSS, regardless of what such markup is. Therefore, let us suppose that the HTML for that piece of content were the shown in figure 14.

I would omit the explanation of some style rules applied to not make this chapter very long, focusing only on the part on which I am most interested. Thus, after applying some style rules to the initial markup, I will take as a starting point for this design exercise the layout that is shown in this figure.

Actually, to accomplish such layout, I have already had to modify the markup shown in figure 14. To achieve the pink border on the right, two extra divs have been added as wrappers of the initial markup:

```
<div id="container">
  <div id="content">
    ... <!-- The initial markup goes here -->
  </div>
</container>
```



Les activités du Club
inscriptions pour le dernier trimestre 2009 auprès du CCAS

Activité yoga	Activité gym douce	Activité gym douce	Atelier informatique	Atelier informatique	Atelier créatif	Atelier couture	Atelier écriture	Généalogie
Lundi	Mardi	Vendredi	Lundi	Mardi	Mardi	Lundi	Mercredi	3ème Vendredi
15h à 16h15	15h à 16h	15h à 16h	9h30 à 11h30	10h à 12h	14h à 17h	14h à 17h	14h30 à 16h30	14h à 17h

proposés les lundis et mardis matins. **Les personnes intéressées peuvent s'inscrire auprès de l'accueil du CCAS** (6 bis rue du Chemin neuf).

L'atelier reprendra ensuite ses cours de septembre 2009 à juin 2010. Les inscriptions sont enregistrées dès à présent.

Les personnes intéressées par ces activités sont invitées à s'inscrire dès que possible auprès du CCAS.

Les activités des cours de gym et des séances de yoga reprendront au sein de l'Espace des Arts et de la Culture dès la mi-septembre pour les seniors biotois.

Le yoga est une discipline qui développe la souplesse et la maîtrise du corps pour une meilleure forme physique et psychique, par des mouvements et postures de relaxation et des exercices respiratoires, afin de mieux lutter contre les stress de la vie quotidienne.

La gymnastique douce propose des exercices d'assouplissement et de tonicité, encadrés par un kinésithérapeute, destinés à améliorer l'équilibre et le forme physique des participants.

La participation aux **activités gym et yoga** s'élève à un montant de **49 € par trimestre.** L'application d'un quotient familial, sur présentation du dernier avis d'imposition, peut permettre une tarification dégressive, sur présentation du dernier avis d'imposition.

Les rencontres du Cercle de généalogie reprendront également de septembre à décembre. Ces rencontres sont animées tous les troisièmes vendredis du mois, par un membre de l'association « Le cercle généalogique de Nice ». Les personnes désireuses d'adhérer à cette association pourront contacter directement **Mme GRIMOULT au 04 93 54 52 10.**

Activités gratuites pour les seniors biotois : les ateliers créatifs, couture, écriture et informatique sont ouverts gratuitement aux seniors biotois sur inscription afin de constituer des groupes.

LES ANIMATIONS

Un séjour dans la région des lacs italiens du 14 au 17 septembre 09 offrira l'occasion de découvrir de magnifiques paysages verdoyants et reflètant la quiétude. Au cours de ce voyage, les participants découvriront tour à tour le Lac majeur et ses îles Borromées : Isola Bella, Isola Madre, l'île des pêcheurs, puis le lac de Lugano, le lac de Côme et enfin le lac d'Orta.
La participation est de 501€ par personne comprenant le transport en autocar grand tourisme à partir de Biot, les excursions et visites, le logement chambre double en hôtel 4 étoiles, les boissons au repas et assurances comprises.

Jeudi 10 septembre 2009 : journée libre sur Entrevaux et sa région
Visite de la cité médiévale en Haute Provence, construite sur un éperon rocheux dont les fortifications sont parcourues par les méandres du Var. Entrevaux fut fortifiée par Vauban sur l'ordre de Louis XIV en 1690. L'accès à la citadelle est assuré de manière originale par 9 rampes en zigzag.

Le village garde de son glorieux passé un patrimoine architectural exceptionnel.

Forfait transport : 8.30 € par personne.

CCAS
«Les Glycines» 6 bis, chemin neuf
Tél. : 04 92 91 59 70
ccas@biot.fr
HORAIRES D'OUVERTURE
des bureaux de 8h30 à 12h30 et de 13h30 à 17h, du lundi au vendredi.

BIOT TERRE DE CENTENAIRES !

Le 9 mai dernier, Thérèse Lacarrière a soufflé ses cent bougies.
Pour l'occasion, Jean-Pierre Dermit, Maire de Biot, était aux côtés de la famille Lacarrière, pour souhaiter un joyeux anniversaire à la deuxième centenaire de la ville (qui rejoint ainsi Alice Carles au rang des doyens biotois).
Plus de quatre générations étaient présentes pour célébrer ce siècle traversé, marqué de grands bouleversements vécus avec courage.

Thérèse Lacarrière et Jean-Pierre Dermit, Maire de Biot.

SOCIAL, HUMANITAIRE, VIE CITOYENNE

15

Figure 13. Page from the 2009 July-August issue of BIOTinfos, the magazine with local information about the village of Biot (Sophia Antipolis, France). The two pink boxes on the half bottom of the page show a relatively simple layout that, however, can not be currently done with CSS.

The outer `div`, `#container`, sets as a background the dark pink colour that is visible in the right column, and a right padding equal to the desired width of such column. It is also relatively positioned to act as a containing block for the absolute positioned `h1` image. As for the `#content` div, it sets its background colour as white and establishes some paddings (`padding: 4em 2em;`). Finally, the width of `#biot-centenaires` have been set, and the image within it floated to the right. There are, of course, quite a few more rules applied, but they simply set the colours and typography of the content, and

```
<h1><img src="social.png" alt="Social, humanitaire, vie
citoyenne"/></h1>
<div id="animations">
  <h2>Les animations</h2>
  <p><strong>Un séjour dans la région des lacs italiens du 14 au 17 septembre
09</strong> offrira l'occasion de découvrir ... </p>
  <p>La participation est de 501 € par personne comprenant ... </p>
  <h3>Jeudi 10 septembre 2009 : journée libre sur Entrevaux <em>et sa
région</em></h3>
  <p>Lorem ipsum dolor sit amet ... </p>
</div>
<div id="biot-centenaires">
  <h2>Biot terre de centenaires !</h2>
  <img src="niza.jpg" alt="niza.jpg"/>
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, ... </p>
</div>
```

Figure 14. **A possible markup for a piece of the content that appears on the page magazine of figure 13.**

change some widths, and I will therefore not mention them here for brevity.

For the first step of the design process, looking at the original magazine page, we could think of floating the last block of content (`#biot-centenaires`), to let the content of the upper block flow around its left side. But that is not how floats work in CSS. If we simply apply `float: right;` to such block, the result is shown in figure 16. That is right because the preceding block, `animations`, does not have an explicit width, so it fills all the available width. Therefore, the floated element can not be placed higher than the bottom edge of `animations`. In addition, by floating the bottom block to the right we have a collateral effect: the right border has been shortened, due to another typical float behaviour: since `#biot-centenaires` is now floated, it is not computed for the height of its container, `content`. This could be easily solved, though, applying one of the clearing techniques that were reviewed in that chapter. And it is

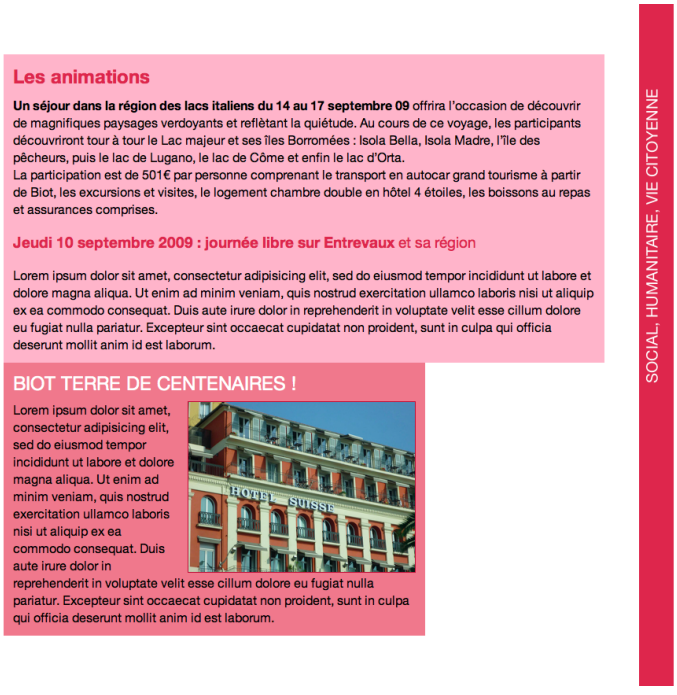


Figure 15. The starting point for recreating the layout of the printed magazine.

not difficult, either, to "pull" its right edge to make it to touch the right border, with the following two rules:

```
#biot-centenaires {
  margin-right: -2.8em;
  padding-right: 3.6em;
}
```

We could then consider to combine the float with some of the other CSS mechanisms that allow to move an element with respect to its current position: **negative margins** and **relative positioning**. Using negative margins, we could try something like this:

```
#biot-centenaires {
  margin-top: -18em;
}
```

However, that gives a result the design shown in figure 17.

Figure 16. After floating the bottom box to the right, this is what is obtained. This is what was expected, since the preceding element takes the 100% of the available width, so that `#biot-centenaires` can not be placed higher than the bottom edge of `animations`.

Relative positioning is not a solution, either, because, as it has already been explained in this thesis, positioned elements are totally removed from the normal flow of the document, so, unless we make room for them, they will overlap or be overlapped by other elements on the page. Thus, if we try to do the following:

```
#biot-centenaires {
  position: relative;
  top: -18em; /* the same than bottom: 18em */
}
```

We would obtain the design shown in figure 18.

Styling a Definition List

The following example is based on an example I first made for one advanced CSS course that I imparted on 2006. After finishing the first part of the course, devoted to XHTML and the importance of first creating a good markup, focusing only on its right *structure* and

Figure 17. Not even using more complex techniques, like applying a negative margin, in addition to the float, to move the box upwards, we achieve the desired layout, and the contents overlap.

Figure 18. Some similar occurs if relative positioning, instead of a negative margin, is used. The only thing that changes respecting figure 17 is that then the content of the top box overlapped the displaced bottom box, and now it is this one that overlap the previous box.

on the semantic of the code, I present the students the screen capture of figure 19, and ask them to do only the XHTML for that page, applying the advices and best practices about structural markup that I have just finished to teach.

For the purposes of this dissertation, though, I am going to focus only on a very specific part of that page, the corresponding to the categories that appear at the bottom of the page: "Alimentación", "Hogar y decoración", "Moda", "Papelería", "Tecnología", and "Otros" (the page is supposed to be a fictitious online store: *Caxigalines*).

The markup that I considered most appropriate for representing those categories is a definition list. It is true that other alternatives would also be valid, such as, for instance, an unordered list with a heading for the title of each category. But let us assume that it has been represented as a definition list, I will show how it can be laid out as appears on the figure. An excerpt of the HTML code for such list of categories is shown in figure 20.

The tough part of laying out a definition list as is shown in figure 19 is that, unlike an unordered list or a sequence of `div` blocks, each logical element of the list is actually compound of two HTML elements: `dt` for the title of the category, and `dd` for the brief description that appears below it. If it had been represented with an unordered list, for example, we could have easily done something like:

```
#categories li {
  width: 33%;
  float: left;
}
#papeleria {
  clear: left;
}
```

The Solution, Step by Step

As summarily as possible, in the next subsections the steps followed until obtain the desired layout are outlined.

Figure 21 shows the starting point from our example: the list without styles. *The List, Unstylish*

Figure 19. Caxigalines.

Fonts and Colours

```
<dl id="categories">
  <dt class="alimentacion">
    <a href="/alimentacion/">Alimentación</a>
    </dt>
    <dd class="alimentacion">
       Ese vino para sushi que te encantó y no encuentras por
       ningún lado, esa especia exótica que te falta para tus
       platos, las mejores latas de mejillones del Cantábrico,
       ...
    </dd>
    <dt class="decoracion">
      <a href="/decoracion/">Hogar y decoración</a>
    </dt>
    <dd class="decoracion">
      Desde la Aluminiun Chair de Vitra y más de dos mil euros
      hasta esos deliciosos tarros de colores para la cocina por
      poco más de diez o incluso esos cuchillos de cocina
      profesionales que no se venden en las tiendas al uso.
      ...
    </dd>
  </dl>
</dl>
```

Figure 20. The list of categories is represented as a definition list in the markup of Caxigalines, the fictitious online store shown in figure 19.

After setting the background and text colours for the different elements, as well as changing some font properties, the list has the appearance that is shown in figure 22.

Setting the Dimensions

In this step, the final dimensions of both the whole list and each of its elements are established. To keep the example as simple as possible, I will use fixed widths and heights: 660 pixels for the whole list, so each category will have 220 pixels (660 ÷ 3). In addition, it is needed to make room for the images of each category (which, since they do not represent actual information, I have chosen to take

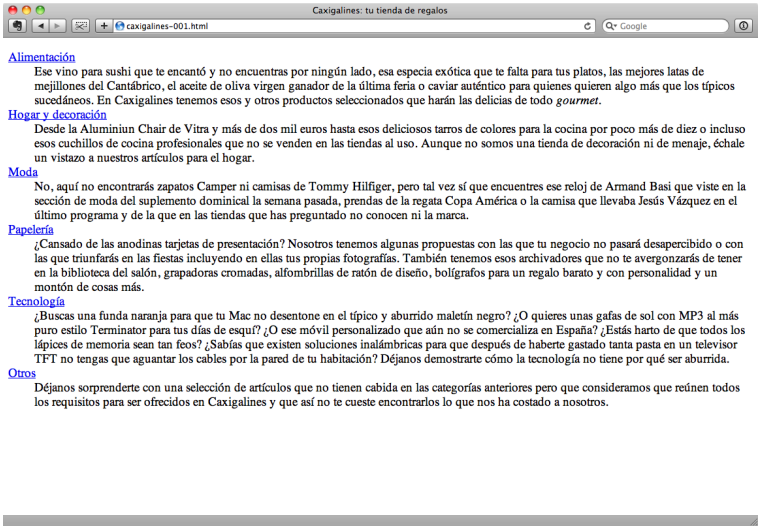


Figure 21. The definition list with no styles applied to it yet.

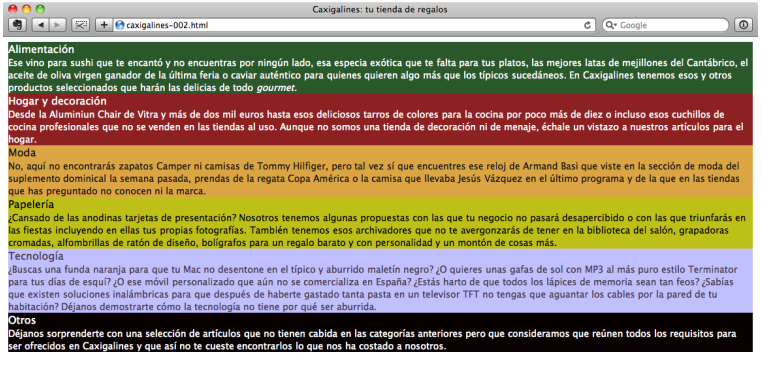


Figure 22. After defining some font properties and the background and text colours for each element of the list, this is the result.

them out of the markup and insert them as background images from CSS). All of that is achieved with the following rules (concrete `background-image` property for each `dt` element have not been included in the code below):

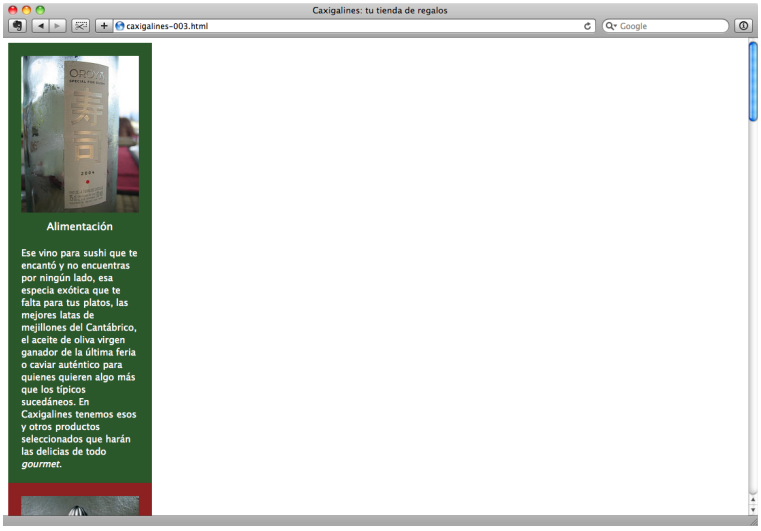


Figure 23. The list, once the dimensions of each category and the background images have been established.

```
#categories { width: 660px; }
dt, dd {
  width: 180px;
  padding: 20px;
}
dt {
  padding-top: 270px;
  background-repeat: no-repeat;
  background-position: 20px 20px;
}
dd { padding-top: 0; }
```

And the result is shown in figure 23.

Floating the List Elements

Once the basic styles have been applied, it is time to begin to lay out the list. The first step is floating all the list elements. As it has already been mentioned, if the categories had been represented in HTML as an unordered list, with this step the layout would be almost finished (only would have to be done a final step to make all categories to be the same height). Being a definition list, where each category is actually comprised of two elements (`dt` and `dd`) the process is a bit

Figure 24. After floating the categories (each `dt` and `dd` element) to the left, this is the appearance of the definition list.

more complicated. Thus, after adding the below rules, we would obtain the layout shown in figure 24.

```
dt, dd {
  float: left;
}
```

Moving the Elements

This is the difficult part of this design: to realise how the final layout can be achieved from the situation shown in figure 24. The key is to apply the same method on which the *One True Layout* technique studied in the previous chapter is based: to combine floats with **negative margins** to move floated elements until they reach their desired position.

In this case, `dd` elements must be placed below their corresponding `dt`, instead of to its right, as they are now. That give us the following needed displacement:

- *Horizontal*: they must be moved the width of `dd` to their left, that is, 220 pixels (width of 180 pixels + 20 pixels of padding for each side).
- *Vertical*: they must be pushed down a distance equal to the height of their `dt`, that is, 310 pixels (270 pixels of padding top plus 20 pixels of padding bottom to the line height of the category title contained in each `dt`).

Applying the following rules, the final layout is almost the definitive (figure 25):

```
dd {
  margin-top: 310px;
  margin-left: -220px;
}
.papeleria {
  clear: left;
}
```

Equal-height Elements

The final step has to deal with a recurrent problem in CSS: making certain elements to be the same height (in this case, category descriptions, that is, `dd` elements). We could think again on the ap-

Figure 25. Category descriptions (`dd` elements) must be displaced below their corresponding `dt`. Naturally, clearing them is not an option, because that would impede the next category to appear vertically aligned with the current one. Therefore, it is needed to apply negative margins to move them downwards and left.

proach that the aforementioned *One True Layout* technique used for this problem:

```
dd {
  margin-bottom: -9999px;
  padding-bottom: 9999px;
}
```

The code above produces the desired final layout, which can be seen in figure 26).

The other option would have been applying the well-known *Faux Columns* technique, using an image with the same six colours than those of the category list, tall enough as to support increasing several times the text size of the browser. However, although in this case this could have been relatively easy, it would have not been possible if the three columns would had not have a fixed size. Note that, if this technique were applied, there would been also necessary that the whole list actually contained its elements. Since all its elements are floated, some of the clearing techniques should be used. Below is one possible code that would work for this other alternative (as long as an appropriate background image were used):

```
#categories {
  background: url(img/categories_bg.png) 50% 50%
              no-repeat;
  float: left;
}
```

Conclusions

From the thirty attendants to the advanced CSS course mentioned at the beginning of this section, all of them employed as web developers, nobody represented the list of categories as a definition list, but as several `div` blocks or, at best, as an unordered list with a heading for the title of each category. Other alternatives were much worse. This is a pattern that is often repeated in my courses: the more experienced is the course, the worse is the markup that they do for this exercise, despite how much I insist on they to ignore the visual appearance of the page once finished, and concentrate only

Figure 26. The final step consists on making all the categories to be (or look like) the same height. This can be done in two different ways: by using the same approach of *One True Layout* technique (setting for each `dd` element an enormous negative bottom margin with the same value for the bottom padding), or applying the classic *Faux Columns* method, defining a background image for the category list.

on the structural markup that they would consider as the best for the information to be represented. Nevertheless, with newcomers to web design, who have never seen anything about CSS before, the results for this exercise are usually much better, and they reach without difficulties a clean and logical HTML.

This is not surprising, tough, and it simply shows how people who are trained on CSS, instinctively tend to be always thinking in *how* that markup could be later laid out, instead of focusing on the structure and semantic of their HTML code. Whilst this is not probably enough to be considered a probatory evidence, in my opinion it does *suggest* that not every layout can be done in CSS regardless of the underlying markup (or, even if possible, it may be very difficult to get done), and this is a lesson that more experienced designers have learned very well.

Vertical Grid

The next case study aims to illustrate how certain layouts, even though feasible, are too complex to be done in CSS, specially when compared how easy it would be using a desktop publishing tool, or even some of the layout languages reviewed on *Chapter 4*. This case study can be considered a mix of the two previous examples: the layout resembles that of the definition list, and the same technique of combining floats and negative margins will be in fact applied to accomplish it; but the dimensions for each piece of content varies, thus involving more changes between the linear order of the source document and its visual layout.

This example is based on an event calendar page from the Spanish magazine *YoDona*, shown in figure 27, weekly distributed with *El Mundo* newspaper. For brevity, I will not show the whole detailed process of how the layout have been performed in CSS, as it was done for the previous examples, but I will merely describe the overall technique followed to accomplish the layout of figure 28.

Since it might be difficult to know what images are associated with what text, specially for a non Spanish-speaking reader, figure 29 shows the distribution of modules in the which, where different

Figure 27. YoDona.

pieces of content (that is, each *event*) are highlighting with different colours.

For this example, I have opted for representing the events as an unordered list, where each event is a `li` element. All the events share the following structure:

- Title of the event
- Author (otherwise, a short tagline)
- Date of the event (day of the week and day of the month)

AGENDA

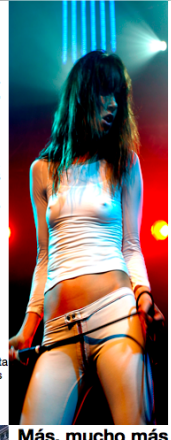
[CINE, MÚSICA, LIBROS, ARTE, ESPECTÁCULOS, DANZA, FOTOGRAFÍA, TEATRO, SOLIDARIDAD]

viernes
13
DIES IRAE
Marta Carrasco
La bailarina y coreógrafa fusiona danza y teatro en esta revisión furiosa e irreverente del Réquiem de Mozart para 15 intérpretes. Hasta el 22 de noviembre en el Festival de Otoño de Madrid. **MÁS INF.:** TEATROABADIA.COM

miércoles
18
LA CINTA BLANCA
Haneke en Segovia
La última película del director austriaco inaugura la IV Muestra de Cine Europeo de la ciudad castellano-leonesa (Muces). El certamen homenajea al director Jaime Chávarri y tiene a Reino Unido como país invitado. **MÁS INF.:** MUCES.ES

*
ALMUDENA BAEZA
Somos/Nos gustaría ser
La artista nos muestra esta instalación sobre la distancia entre la realidad y el deseo en el espacio de arte Frágil (Madrid). La pieza se extiende en forma de caálogo (a la dcha.), en edición limitada **MÁS INF.:** FRAGILESESPACIODEARTE.BL

miércoles
25
JULIETTE LEWIS
Salvaje rockstar
Con la carrera como cantante viento en popa, Lewis vuelve a sudar y desmelenarse con su aquelarre de rock primitivo y arañazos punk. En la Sala Heineken (Madrid). **MÁS INF. Y FECHAS:** TICKETMASTER.ES



LA BARCELONA CANALLA
Màrius Carol
El periodista y escritor echa la vista al siglo XX para desvelar, en *Las plumas del marabú*, la intrahistoria erótica y sentimental de una Barcelona capital inesperada de la *dolce vita* europea. Publica La Esfera de los Libros. **MÁS INF.:** ESFERALIBROS.COM

viernes
20
LOS CONDENADOS
Bárbara Lennie
La actriz demuestra sus dotes dramáticas en la última cinta de Isaki Lacuesta (*Cravan vs. Cravan*), una denuncia de los traumas dejados por las dictaduras en Latinoamérica, en relación con los desaparecidos y su memoria. **MÁS INF.:** BARTONFILMS.ES

lunes
16
MAISON MUMM
Cultura en Burbujas
La bodega GH Mumm reproduce en el hotel AC Santo Mauro (Madrid) el *château* de origen en Reims, para desplegar toda su sabiduría en torno al mundo del champán, con una exposición y tienda, ediciones exlusivas, catas, diálogos con sumillers… Hasta el 21 de noviembre. **MÁS INF.:** AC-HOTELS.COM

Más, mucho más
José Carlos Plaza
Bodas de sangre
El director subraya la fatalidad y la sexualidad de esta pieza fundamental en la obra de Lorca. Hasta el 3 de enero en el Teatro María Guerrero (Madrid). **MÁS INF.:** CDN.MCU.ES

Figure 28. YoDona.

Figure 29. YoDona.

- An image representative of the event, intended to attract the reader's attention
- A short description
- Where to get more info about this event (a web site or telephone number)

Following there is an example of how a single event is modelled in HTML:

```
<li id="diesirae">
  <h3><cite>Dies Irae</cite></h3>
  <h4>Marta Carrasco</h4>
  <p class="date">
          <span class="weekday">Viernes</span> <span
class="day">13</span>
  </p>
  <p class="description">
    <img src="images/diesirae.jpg" alt="Escena de
«Dies Irae»"/> La bailarina y coreógrafa fusiona danza
y teatro en esta revisión furiosa e irreverente del
Réquiem de Mozart para 15 intérpretes. Hasta el 22 de
noviembre en el Festival de Otoño de Madrid. <span
class="moreinfo">Más <abbr
title="información">inf.</abbr>: <a
href="http://teatroabadia.com/
">teatroabadia.com</a></p>
</li>
```

In this case, the grid used by this page of the magazine is obvious: 4 rows × 5 columns. Every column has the same width, but rows are of different height. Specifically, first and second rows are equal height, the third one is shorter, and the fourth row is larger than the two first ones. Let us see how the unordered list can be laid out to mimic the design of the magazine.

For this example I will use again a fixed width. Specifically, I am defining a width of 900 pixels for the calendar (the whole list), thus obtaining five columns of **180 pixels**.

General Styles

To accomplish the layout, first the following general styles are first applied:

```
#agenda li {
  width: 170px;
  padding-top: 65px;
  min-height: 235px;
  float: left;
  position: relative;
}
#agenda img {
  width: 180px;
  height: 100%;
  position: absolute;
  top: 0;
}
#agenda .date {
  position: absolute;
  top: 0;
}
```

Each `li` element is first set a **`width`** of **170 pixels**. I have did it so because later I will leave a space between the text and the image for each event of 10 pixels. CSS does not allow to define anything similar to the `rowspan` or `colspan` of HTML tables, so the *modules* must be simulated, computing vertical and horizontal dimensions and displacements for each element in the list.

The **`padding-top`** of 65 pixels is for leave room for the date, that must be absolute positioned at the top of each "cell", because it is not defined in that position in the HTML, and therefore absolute positioning have to be used (it is exactly the same situation that happened in the first of the case studies of this chapter). For that reason, `li` must have a value for their `position` property other than `static`, so that they can act as containing blocks for the absolute positioned date inside.

Finally, each list element is floated to the left.

In addition, all images are set a width of 180 pixels, and a height of 100% (that is, they will fit the height of its containing `li`). Although this is far from being a good practice (setting both the width and the height of images and any other replaced element, because

by doing so we are altering their proportions), if each image is carefully chosen of a presumably similar ratio to their final aspect, the distortion may be not excessively noticeable (and we have no other option if we want a perfectly vertically aligned grid like the one used in the magazine).

Changing the Dimensions of each Module

Now all list elements must be sized according to the module they represent in the grid. Despite this is not the most difficult part of this design exercise, it is a tedious process, specially in a grid like this where there are several modules of different sizes and orientation. Put differently, what we are doing in this step is emulating what `rowspan` and `colspan` attributes would have been done if we were laying out this design with HTML tables.

To not make this chapter longer than it is, I only depict how it would be for two types of modules.

Two Columns Module with the Image on the Left

Let us consider, for example, the first item on the top left position of the grid: *Dies Irae*. Remember that each `li` has been given a width of 170 pixels; however, since this event spans two columns, its final width would be 360 pixels (two columns of 180 pixels), to make room for the image on the left. This can easily be done with the following code:

```
li#diesirae {
  padding-left: 190px;
}
li#diesirae img {
  left: 0;
}
```

The actual width has been augmented 190 pixels width the left padding applied, and therefore it will be equals to 170 + 190 = 360 pixels, leaving a 10 pixel separation between the image and the text on the right.

In addition, the image (that has already been absolute positioned on the top of its containing `li` in the previous step), is now positioned on the left edge of its container.

One Columns and Two Rows, with the Image on the Top

Another very different module is that on the top right corner: *La Barcelona canalla*. This time is a single-column module that spans the first and second row of the grid. We have to apply to it a left padding of 10 pixels, to separate the text from the other elements on its left, and then comes the difficult part: the description of the event must be moved downwards so that its image (*Las plumas del Marabú*) can be placed on top of it.

Naturally, this would be very easy if we simply applied the same method than we did for the previous module, that is, define a padding (in this case, a `padding-top`). But, what padding to apply? Because, unlike the previous `li`, now we have to define a *vertical* padding, instead of a horizontal one, and therefore we encounter the usual problem in CSS: every time that an explicit height is set, the layout suffers when not every factor is as we planned (browser text size, screen resolution, dimensions of the browser window, installed fonts…).

Other options would be:

- Defining a margin-top
- Using relative positioning to move the `li` downwards

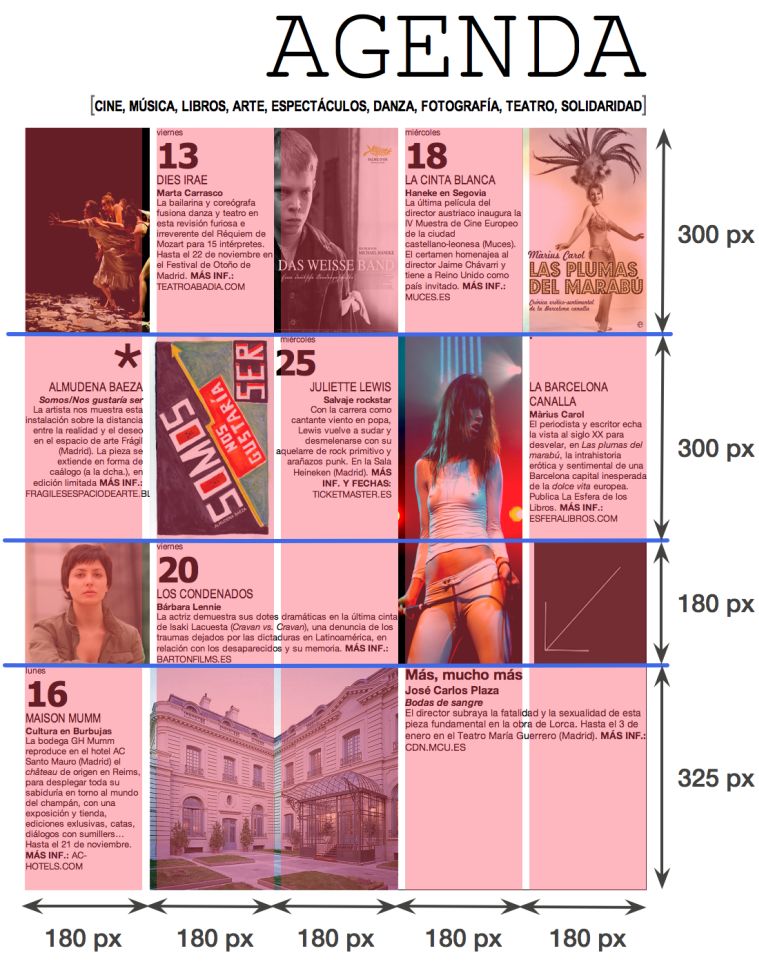
Each have their advantages and drawbacks, and depending on what option is finally chosen, the rest of the steps will also vary accordingly. However, both alternatives have exactly the same problem than before: it is needed to know the height.

To sum up, *it is not currently possible to do a layout like this in CSS without setting an explicit height for the rows of the grid*. In other words, there is no way to define the height of one element in relation to another one.

Therefore, if we want to create this design with CSS, a decision must be taken for the heights of the rows. In order to be able to continue and complete the example, I will make the rows to have a height of 300, 300, 180, and 325 pixels, respectively, as it is depicted on figure 30.

Once the height of the rows have been established, any of the three aforementioned alternatives would be feasible for this module. For example:

Figure 30. YoDona.




```
li#marabu {
  padding-left: 10px;
  padding-top: 300px;
}

li#marabu img {
  left: 0;
}
```

Moving the Elements

After resizing each element as it has been done for the two previous examples, would begin the really tough part of this layout: due to the complex rules that govern floats, each element has to bee "manually" moved to its final position, in a similar way to what was done in the section . Since it would be very verbose to explain here the whole process, and it depends, as it has been stated before, on the previous strategy followed for changing the dimensions of each element, I will simply show one of the multiple possible solutions for this design (see figure 31).

Conclusions

CSS layouts tend to have a visual structure that matches the linear order of the document, which limits the designers' creativity. Of course, in theory it is possible to alter such order, and that is for which absolute positioning is intended. In practice, though, as we have seen for the title and date of blog posts, and the headlines and summary of news articles in a newspaper, even minor changes in the layout that mismatch the logical structure of the document are difficult to achieve with guarantees: unless we know exactly how much vertical space must be left between the title and the top edge of its container, absolute positioning is not applicable. And, although it is possible to reach to a compromise solution that works reasonably well and resists a few changes in the text size, this is no longer true if we can not know a priori how many lines of text expands the absolute positioned element. For the case of dates, this may not be so problematic, but that is not the general case, as it has been shown in the newspaper example: liquid layouts, mobile devices, and unpredictable text, as is the case of the majority of current websites, run by content management systems, limits this technique, making absolute positioning almost impracticable.

Other layouts are, however, very difficult to achieve, if more severe changes between the structure of the document and the visual layout have to be done. Vertical alignment among elements is simply not possible, unless we define a fixed height for the involved

```
#agenda li {
  float: left; width: 170px;
  min-height: 235px;
  padding: 65px 0 0 190px;
  position: relative;
}
#agenda img {
  width: 180px; height: 100%;
  position: absolute;
  left: 0; top: 0;
}
#agenda .date {
  position: absolute; top: 0;
}
li#marabu {
  position: relative; top: 300px;
  padding-left: 10px;
}
li#marabu img { margin-top: -300px; }
li#juliettelewis {
  margin-right: 180px;
}
li#almudenabaeza,
li#juliettelewis {
  padding-left: 0;
  padding-right: 190px;
  text-align: right;
}
li#almudenabaeza img,
li#juliettelewis img {
  margin-left: 180px;
}
li#juliettelewis {
  min-height: 415px;
}
li#juliettelewis img { height: 480px; }
li#loscondenados {
  margin-top: -180px;
  min-height: 115px;
  width: 350px;
}
li#maisonmumm {
  padding-left: 0;
  padding-right: 370px;
  height: 260px;
}
li#maisonmumm img {
  width: 360px;
  margin-left: 180px;
}
#muchomas {
  float: left;
  width: 349px;
  min-height: 324px;
  margin-top: -180px;
  padding-top: 180px;
  padding-left: 10px;
  background: url(img/arrow.png) right
              top no-repeat;
  border-right: 1px solid black;
  border-bottom: 1px solid black;
}
```

Figure 31. **A possible solution for the layout of *YoDona* magazine, defining explicit "row" heights and using a mix of floats, negative margins, and absolute and relative positioning.**

elements. But then, as it has been shown, we would run into the same problems than absolute positioning.

This is one of the reasons why is so difficult to find examples of web sites where the design changes drastically from one pages to others, something that is, however, very common in printed magazines.

8

The Problem of Separation between Structure and Layout

This chapter is the end of the first part of this dissertation, which have been devoted to demonstrate the hypothesis that opened this thesis: CSS is not a layout language. Previous chapters have revealed the major issues of floats and absolute positioning and their inadequacy as layout mechanisms. The more advanced layout techniques and other approaches also studied in previous chapters can not be considered a solution to the problem of layout on the web, either. As a consequence, separation between content and presentation or, more specifically, between structure and layout, is not possible with Cascading Style Sheets in its current version.

Introduction

Separation of presentation and content is probably the most common feature attributed to Cascading Style Sheets, so often repeated that is has became a commonplace. Nevertheless, it is far from being true, specially with respect to layout. Whereas style sheets have had an enormous success removing `font` tags and other presentational elements from the markup, there still are many sites that continue using HTML tables for layout. And even those others with *pure* CSS layouts usually have a markup that is dependent, to a greater or lesser extent, on their final visual layout.

Separation between presentation and content in CSS is not longer true when it comes to layout.

The title of this chapter is not therefore fortuitous, but it has been meticulously chosen to reflect that Cascading Style Sheets calls for a further degree of separation to distinguish the low level typographic aspects of a document, such as font size and type, list styles, colours, etcetera, for which the technology works particularly well, from those other high level layout issues, such as the number of columns of a document or the vertical alignment among different elements on the page, an area in which Cascading Style Sheets is still very immature.

Thus, notwithstanding the widespread use of the *separation between presentation and content* expression, and to help focusing on the specific problem tackled by this thesis, I propose to divide the term presentation into two different components: *style* and *layout* (see figure 1). Whilst the former refers to the aforementioned stylistic effects that were traditionally accomplished by `font` and other physical elements and attributes, by **layout** I mean the arrangement of the elements on the screen or paper when the document is visually rendered.

On the other hand, although both "content" and "structure" are often indistinguishably used to refer to the (X)HTML document, I have favoured here the use of *structure* over *content* to emphasise the fact that the lack of true layout mechanisms in CSS is usually forcing authors to change the markup to obtain the desired layout, either by adding superfluous markup —most often in the form of non struc-

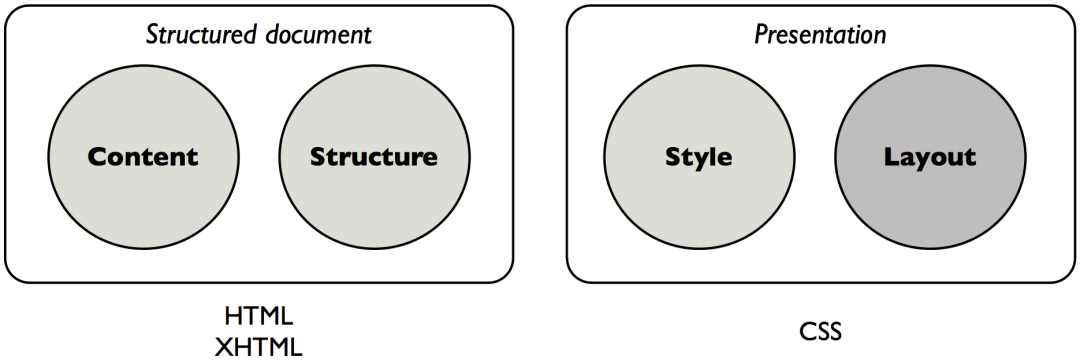


Figure 1. The figure shows a more detailed separation of concerns than the oft-expressed content and presentation. Although HTML represents well both the content and the structure of the document, and thus both terms can be used without distinction, this thesis claims that presentation includes both style and layout. While CSS allows to change most stylistic aspects of a document independently of its markup, the same is not true when applied to layout, breaking thus the promised separation between content and presentation, or, more specifically, between structure and layout.

tural *divs* and classes or identifiers, which has been known as *classitis* and *divitis* (Zeldman, 2003)— or altering the logical structure of the document to convey its visual representation.

CSS Is Not a Layout Language

This is the hypothesis upon which this thesis rests, expressed in its concisest form: Cascading Style Sheets is not a layout language. Despite it is, with no doubt, a bold affirmation, given the nicely designed web sites that can be seen nowadays, entirely made with standard (X)HTML and CSS, I consider it has been sufficiently demonstrated in previous chapters that CSS, as we know it today, is inadequate for carrying out the layouts that web designers are demanding: not only there are certain tasks that can not get done with CSS; many others can only be accomplished adding extra markup or altering the logical order of the content. And all of them at the cost of being extremely complex when compared to what could be done with HTML-table based layouts.

This must no be understood, though, as a criticism to the inventors of Cascading Style Sheets. When the first specification of Cascading Style Sheets appeared (Lie & Bos , 1996), they simply could not anticipate the way in which it is used today. This has a simple explanation: the resolution of computer screens. By the time CSS1 was developed, 640 × 480 and 800 × 600 were the most common resolutions of computer screens (see table 1).

Table 1. Computer display resolution in 1997 and 1999, as it appears in Nielsen (2000, p. 28). Data for 1997 are calculated based on the results of 5,000 users who accessed www.horus.com, and more than 11,000 participants in the GVU survey. Data for 1999 come from www.statmarket.com.

Screen resolution	1997	1999
Very small (640 × 480)	22%	13%
Small (800 × 600)	47%	55%
Medium (1024 × 768)	25%	25%
Big (1280 × 1024 or greater)	6%	2%

Computer display resolution in 1997 and 1999

With those screen resolutions, multicolumn and grid layouts were so impracticable that most web sites were just a flow of text with just one column, like the example shown in figure 2. Thus, the only scenario they contemplated was the ability to float an image or a navigation menu to one side of the page and let the content flow around it, something that were possible in most word processors and, of course, in publishing tools.

Anyway, the fact is that CSS was born without layout capabilities, and despite the addition of positioning mechanisms to CSS2 (Bos, Lie, Lilley & Jacobs, 1998), and the efforts of web designers to deal with such a minimal equipment, pushing CSS to its limits, it still suffers from the same lack of true layout mechanisms.

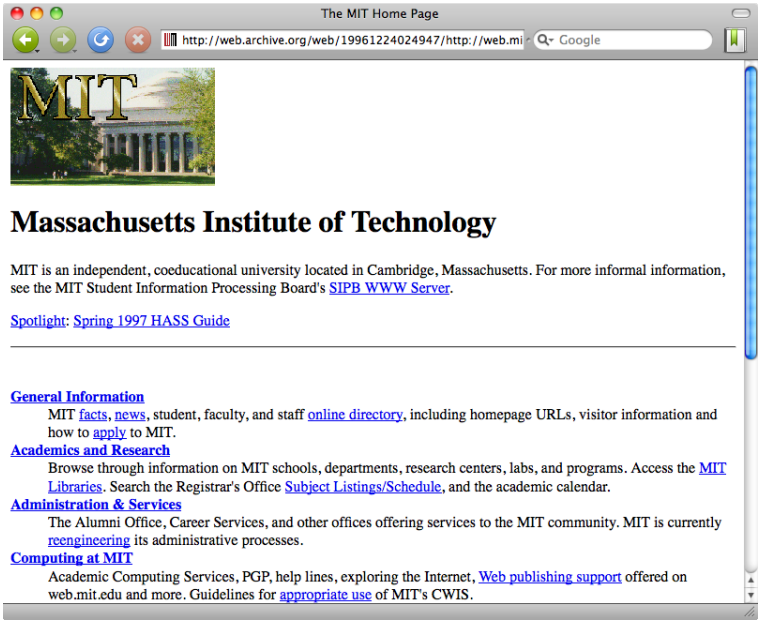


Figure 2. The picture shows a screen capture of the web site of Massachusetts Institute of Technology (MIT), as it was shown in 1996 (extracted from Web Archive: `http://web.archive.org/web/19961224024947/http://web.mit.edu/`).

Next sections briefly describe each of the problems of Cascading Style Sheets in respect to layout that have been identified through the course of this research, and that have already been demonstrated in the previous chapters.

The Problem with Floats

Despite the widespread use of floats as a layout mechanism, they were never conceived for that, but as a tool for letting the content flow around a floated element.

As it has been stated in *a previous chapter, floats were never intended as a page layout tool*, but as a way of allowing the text to flow around the side of an element (typically, an image). Nevertheless, today it is the most common technique for doing multi-column layouts with CSS. There is a pragmatic reason for this use, to wit, the floats ability to be cleared, which permits a footer, for example, to be put below the preceding columns (Meyer, 2004c), *something that it is not possible with absolute positioning*.

But, although it is possible to use them to achieve certain layouts, their use presents several problems which are discussed below.

Dependent on the Order of the Content

The principal inconvenience of using floats for layout purposes is their dependency on the order of the content in the HTML docu-

ment. This is due to the restriction, imposed by CSS 2.1 specification, that the outer top of a floating box may not be higher than the outer top of any block or floated box generated by an element earlier in the source document, nor than the top of any line-box containing a box generated by an element earlier in the source document (Bos, Çelik, Hickson & Lie, 2009, §9.5.1). In other words, what the specification is saying in the constraints five and six of the cited section, is that a floated element may not move upwards than any other preceding element in the source code (be it floated or not).

Analogous rules exist that govern the behaviour of floats with respect to their horizontal position. Although Robinson (2005) has demonstrated that it is possible to achieve any number of columns in any order, his technique, known as its seminal article, One True Layout, which involves not only floats but also negative margins, and requires a precise understanding of the box and visual formatting models, is beyond the comprehension of the average designer. As Baron (2006) has stated, "these techniques are extremely complex, fragile, and hard for authors to write".

It is the Content, and Not the Box, What Flows

Even in what could be considered a more standard use of this property, such as aligning an element to the left or the right side of its container and letting the content flow around its opposite side, the use of `float` is not exempt from problems either. Let us think, for example, of a menu aligned to the left of the page (a simple layout that was quite common a few years ago). If a background were applied to a block element of the surrounding, it would expand beneath the floating menu, as figure 3 is showing. This is due to another feature of floats that have already been explained in a previous chapter: *it is the content, and not box of the element itself, what flows*.

The Problem with Absolute Positioning

Positioning did not form part of the first specification of CSS (Lie & Bos , 1996). It was publicly announced as a Working Draft (Furman & Isaacs, 1997) on the www-style@w3.org mailing list on January 31, 1997, and later added to the CSS2 specification (Bos, Lie, Lilley & Jacobs, 1998). According to the first draft, the inability to explicitly control the positions of HTML elements had become a barrier

Figure 3. When a element, such as a menu, is floated, the content of the following elements flows around it, but not the boxes themselves. Therefore, if these elements have a background colour or a border, they visually go beneath the floated element (as it is shown in the figure on the left). Thus, a layout like the shown on the right figure is not currently possible in CSS.

to producing rich static HTML documents. Thus, it was born as a way "to allow authors to exercise greater accuracy in page description and layout".

The ability of absolute positioning to take an element out of the normal flow of the page is also its major inconvenience.

The major strength of CSS positioning is its ability to take an element out of the normal flow of the page, so it allows, in theory, to place an element at whatever position on the page. But this has became its main problem, too. Since the user agent does not longer care about the positioned element, the author is the responsible for control that it does not overlap the rest of elements on the page. While this might not be a problem when it is used for column layouts, since it is often possible to specify margins or paddings so that there is room left for the positioned element, this is not usually the case when we want to move the element vertically on the page. Unless we know exactly the height of the positioned element, it is not possible to use absolute positioning for altering the order of the content when it is visually rendered. Although most modern browsers have a very improved zooming feature which works reasonably well and they are even able to adapt absolute positioned elements when the user increases the size of the font, it is still possible they overlap when the browser window is resized, unless the the absolute positioned elements (or their containers) have an explicit width or height. Anyway, this feature of some browsers, although represents

a significant accessibility improvement, is not sanctioned by the specification, so we should not rely on it.

In any case, even using fixed layouts, it is needed to know a priori the content of the positioned element to know how much space the rest of elements of the page must leave to accommodate it, so absolute positioning is not usually a technique that can be used, for example, in a Content Management System template, where the actual content is left to the final user instead of the designer. Some authors have trying to get the best of both worlds (the independence of absolute positioning from the order of the content without removing the element from the normal flow of the document), using techniques such us Faux Absolute Positioning (Sol, 2008), but, once more, at the cost of complexity (and, in the specific case of this technique, extraneous markup too).

A Note on Page Zooming and Absolute Positioning

Nowadays, with the improved zooming capabilities of browsers, an absolute positioned layout behaves reasonably well under changes in the size of the text (since the default behaviour of browsers tends to be make zoom and not just increase the font size). Of course, this remains a problem in the case of liquid layouts.

It is true that various recent developments make the height of elements a little bit more predictable:

- The page zoom function distorts pages less than the text zoom.
- Downloadable fonts avoid uncertainty about which fonts are used
- The W3C CSS Working Group is discussing a property that scales a font to fit a box instead of the other way round, and a similar feature is also being considered for images to adapt to their containing block.

But there is still no guarantee that browsers justify text and break lines in the same way, even if they use the same downloaded font. And the height is still only (*somewhat*) predictable if the width is specified in ems. And even then the zoom function only keeps the aspect ratio of boxes the same if the width does not depend on the width of the window.

There is the problem of dynamic content, too: if an author wants to use the same style sheet for multiple pages, or for transla-

tions of a page, the height of the text will be different. For instance, if we want two boxes of text to have the same height, we could try to size the boxes such that they have enough room if the text is written in Italian (or some other language that uses a lot of space). But then if the text is in English or Chinese, there would be a lot of wasted space. It would be much easier if the renderer itself could determine the height of both boxes and make them *exactly* the height of the longer of the two. That is one of the reasons why tables are still used, despite the disadvantage of having to change the document source when the design changes.

Vertical Grids Are Not Possible

If multi-column layouts are not an easy task in CSS, vertical alignment of the elements on the page are simply not possible (at least, not without recurring to methods such as adding non structural elements to act as containers and floating them and their children —which, again, is dependent on the order of the content—, using JavaScript to calculate the height of the elements or using background images). Decidedly, there should be a simpler manner to specify, for example, that two columns must have the same height or that a certain element must be as tall as the sum of other two, something that was trivial using HTML tables.

Mixing Units Is Not Possible

Although this is not a so severe problem like the others mentioned so far, it represents another inconvenience of CSS for layouts: the inability to mix several length units. One typical example is that of a two or three column layout where column size has been defined, let us say, in percentages, and we want to assign a padding to each column of `1em`. Despite there is, of course, technically possible, by doing so we lose every control over the dimensions of the global layout (we could not make it to adjust to the total available width, since there is no manner to say that the width of a column must be, for example, `33% - 2em`). And even in the case that the size of each column were defined in `em` (assuming that they form an elastic lay-

out), it would require recalculate margins and paddings every time a `font-size` were applied to a whole column.

For this reason, some authors recommend to add an extra `div` to every container (Cederholm, 2008, p. 218), to serve as a placeholder where to apply margins or paddings in any unit without affecting the overall layout (since those properties are applied to a child element, they do not modify the dimensions of its container).

Extra Markup Is Usually Needed

As it has been shown, most layout techniques involve adding extra, non semantic markup, to obtain the desired effect.

There Are Not Content Reordering Mechanisms

One of the missions of graphic designers is to visually present the content in the most appropriate manner to effectively convey its message. This involves not only playing with fonts and colours, but also to arrange the images and text of the original document to adapt them to the medium in which they are being presented. Frequently, this does not match what will be the logical order of the content if it were displayed as a flow document.

CSS does not provide a mechanism to make this reordering possible. Elements can not be placed related to other documents on the page. This is because Cascading Style Sheets has inherited the notion of flow text documents, as opposed to layout documents such as magazines, posters and brochures. In other words, it is not currently possible to do with CSS what traditional publishing tools like QuarkXPress, Adobe InDesign, Adobe FrameMaker, etcetera, have let designers do since many years ago (and that even the latest version of Apple Pages allows now for domestic users too).

Redesign Is Not Possible

The most clear evidence of the lack of true layout mechanisms in CSS is when it comes to redesign, that is, when we want to completely change the design of a page. Whereas, if separation between presentation and content were true, this should be as easy as to indicate a different style sheet to the document, without altering the

HTML document, this is almost never the case. Except for changes in the font families, colours and background images, etcetera, whenever redesign also involves drastic changes in the layout, it usually implies adding extra markup, altering the logical order of the content, or both. Thus, this thesis states that redesign is not currently possible with Cascading Style Sheets without breaking the separation between presentation and content.

Is It not Possible? What Does It Happen with CSS Zen Garden?

Last sentence is certainly bold, especially if one thinks of CSS Zen Garden. This well-know web site, launched in 2003 to promote the use of CSS (its motto is "a demonstration of what can be accomplished visually through CSS-based design"), now holds 210 official designs —more than one thousand submitted in total—, each one comprising just one style sheet (and its corresponding background images) that modify the same HTML document.

Although those designs are apparently very different among them, with layouts that involves one, two, or three columns, some others which expands horizontally instead of vertically, etcetera, if we carefully examine all the designs, what initially looked like radical different designs ends up being just variations of a few common layouts. Most of the changes are achieved through an intensive use of image replacement techniques, background images and fine typography. But the underlying layout remains almost the same. There are very few designs that change the order of the different sections of the page, for example. And, when they do, they usually limit to move some paragraph to the top or bottom of the page, using absolute positioning with fixed layouts, something that makes them fragile to font size changes and prevents them to adapt gracefully to small devices, like mobile phones.

Moreover, in order to make such changes possible, the underlying markup is anything but clean: many superfluous, non-semantic elements, with a huge amount of classes and identifiers are provided to give designers the needed *hooks* for style being applied.

Thus, despite the contribution that CSS Zen Garden has made to widespread the use of Cascading Style Sheets, it can not be considered a refutation of the statement that opened this section. I have

therefore to disagree with the following sentence that appears in the content of the CSS Zen Garden page:

> *CSS allows complete and total control over the style of a hypertext document.*

As far as layout is concerned, the above quoted paragraph is false.

Complexity

CSS is becoming a more complex language than it was intended to be. This is especially true when referred to layout. Final users should not have to be aware of the verbose rules, constraints, interdependencies, exceptions, and exceptions to the exceptions, that govern floats, absolute positioning, collapsing margins, and the box model and visual formatting model in general.

Lack of Visual Tools

Related with the problem of complexity, it is the lack of visual tools for creating advanced CSS layouts in a WYSIWYG way. Despite this type of applications exist almost since the appearance of Cascading Style Sheets, and though they have improved in recent years, they are still very far from the high level of abstraction that is offered by their corresponding desktop publishing tools. All of them are limited, to a greater or lesser extent, to provide a visual interface for editing the underlying CSS rules and properties, but there is no way to tell them, for example, that an element must be as high as the sum of other two, or that it must be placed below other element. At most, what they are commencing to incorporate are *templates*, that is, a set of predefined, and more or less configurable, layouts, from which the author can select the desired one for the document being edited.

This is not anything but, again, a demonstration of the statement of this thesis: although the existence of this sort of tools could alleviate the complexity of creating layouts with CSS, it is indeed that complexity what prevents more advanced tools to exist. The gap that currently exist between the constraints that could specify a layout at the desired high level of abstraction, and the low-level

properties with which they have to be created in CSS, is so huge that it can not be automatised by a tool.

Conclusions

As this thesis has demonstrated, Cascading Style Sheets lacks real layout mechanisms, which prevents a true separation between presentation and content. Floats were never intended as a layout tool, and absolute positioning, although very flexible in theory, is only practical for very concrete situations, where the dimensions of the positioned element are under the control of the designer, so the risk of overlapping can be avoided. As a result, both mechanisms are not suitable for specifying the overall layout of a document. Moreover, they often require superfluous markup, or, what is even worse, to alter the logical order of the content.

Some authors have proved that these properties can be used in more imaginative ways, combining them with other features of CSS, like negative margins, to accomplish a greater level of flexibility and therefore more independence between the structure of the document and its visual layout. But this is at the cost of an extreme complexity, beyond the understanding of the average user (if they were even conscious of their existence). And even in the case of experienced designers who are aware of such techniques and are able to comprehend them, these are so difficult to implement that most of the times they end up relaying on more traditional uses of floats and absolute positioning, despite it leads to a not so clean, structured, and ordered HTML. Therefore, these techniques, notwithstanding their undoubtedly merit, serve more as an experiment of what it is possible to be done with CSS than as actual solutions to be daily used.

But these techniques are telling us something. If web designers are making such efforts to achieve different ways to specify layouts, it is because there is a hole in Cascading Style Sheets as we know it today. Why is nobody attempting to do the same for changing the colour of the text or the font size of an element? Because CSS

What we need is to be able to specify the layout in a explicit way, instead of having to deal with low-level mechanisms such as floats and absolute positioning.

already works very well for these sort of things. The problem, as this thesis states, is that we have no way of defining the layout of a document **explicitly**. Instead, users are forced to do it implicitly, dealing with low-level mechanisms —namely, floats and absolute positioning— which lack the needed expressiveness to to accomplish the layouts that web designers are demanding nowadays. Making an analogy with software engineering, I would say that it is like we were still developing any complex modern application in Assembly language. Is it possible? Sure (at least, in theory: after all, everything is to be machine code). But, is it *actually* possible? That is, could a C++ compiler, a web browser, or an e-commerce application have been programmed entirely in that way? Certainly not. The amount of time required to deal with such a complexity would make it a Herculean effort, and these kind of applications would have been probably never conceived if high level programming languages had not appeared. Similarly, there are certain tasks that are simply impossible to achieve with CSS. Vertical grids are decidedly an issue, as is the lack of content reordering mechanisms.

It is therefore necessary to provide Cascading Style Sheets with mechanisms that allow us to define explicitly the layout of the document at a higher level of abstraction, something that designers have been doing since the invention of grid systems, and which is not yet possible on the web.

9

Proposed Solution: The CSS3 Template Layout Module

This chapter describes thoroughly the proposed solution to the problem of layout on the web. It is a proposal of addition to CSS, intended to be included in a future specification of the standard. The solution presented here mostly corresponds with the CSS3 Template Layout, an official W3C Working Draft, of which the author of this thesis is one of its editors.

This chapter does not limit to be a transcription of the current Working Draft, though, but explains the motivation for some design decisions that have been taken and discusses some alternatives. In addition, some extensions that have not yet been accepted to be part of the module, but that, in this author's opinion, would be desirable, are also presented here.

Redde Caesari, quae sunt Caesaris

It is complicated to present as the proposed solution of a thesis something that has been developed —that it *is* being developed— within the W3C CSS Working Group[1] (CSS-WG), since it is never the one's own work exclusively, but the result of a joint effort of the rest of the members of the group, as well as the community that contributes to it through the public mailing lists, attendants of the conferences and workshops where it has been presented, opinions that other web designers and CSS experts have expressed in their blogs, etcetera.

And it is even more difficult when, in case of having to award the credit to one person, that person would certainly be **Bert Bos**, who had already written a first draft of this module when I joined the CSS-WG. Be it for him, thence, my public recognition in these lines as the original inventor of the module presented here, of which I have only been a contributor.

About the Solution

One of the novelties in CSS3 is that it will no longer be a single monolithic document, but is divided into a set of separate modules. The solution presented in this thesis to the problem of layout on the web is one of such modules, the *Template Layout Module*, a W3C Working Draft, formerly known as *Advanced Layout Module*, coauthored by this thesis' author and one of his supervisors.

Being a Working Drat means that it may be modified or even discarded as a part of CSS3 in case of failure to reach consensus among the W3C CSS Working Group members. In any case, the syntax and features that are described here correspond to those of the version of 2 April 2009, which can be found on this permanent address: `www.w3.org/TR/2009/WD-css3-layout-20090402`. The latest version is always available on `www.w3.org/TR/css3-layout`.

1 www.w3.org/Style/CSS/members

Introduction to the Template Layout Module

As it has been repeatedly argued in this dissertation, it is necessary to count with a high-level mechanism that allows to define the layout in a explicit manner. And it should also be possible to position elements in whatever area of the page regardless of their actual position in the source document. To meet these requirements, the proposed solution is built on two main concepts: *templates* and *slots*.

The Template Layout Module was conceived with two main motivations in mind:
a) to define the layout explicitly, in terms of rows and columns and the relationships among them
b) to provide a content-independent positioning scheme

A **template** is a two-dimensional structure of *rows* and *columns*. Current CSS mechanisms for layout, such as floats, relative and absolute positioning, negative margins, etcetera, operates over the specific elements to be positioned, and the resultant layout is implicitly defined by the complex interactions among all those low-level mechanisms that are applied to single elements. Conversely, a template explicitly defines the layout of the element to which it is applied (which can be the whole page if the template is defined in the `body` element).

Instead of changing the position of the element to which it is applied, a template sets a *layout policy* for its contents.

In other words, a template does not alter the position of the element where it is defined. Instead, it defines certain regions, or *slots*, where the contents of the element are to be positioned. And it does explicitly, at a high level of abstraction, in terms of rows and columns, and the relationships among them, very similar to an HTML table, but without establishing any binding between the content and the template. The binding is performed in a second stage, where concrete elements are positioned into the slots defined by the template.

A template is made up of **slots**, which can span several rows and columns, according to the syntax and rules that are explained in the following section. Slots represent regions where the contents of the template element can be positioned, regardless of their actual order in the source code of the document. This is shown in figure 1, which depicts a template made up of four rows and three columns that in turn define six slots (named from *a* to *f* in the figure).

That layout could have been obtained with the following template (the concrete syntax is detailed in the following section):

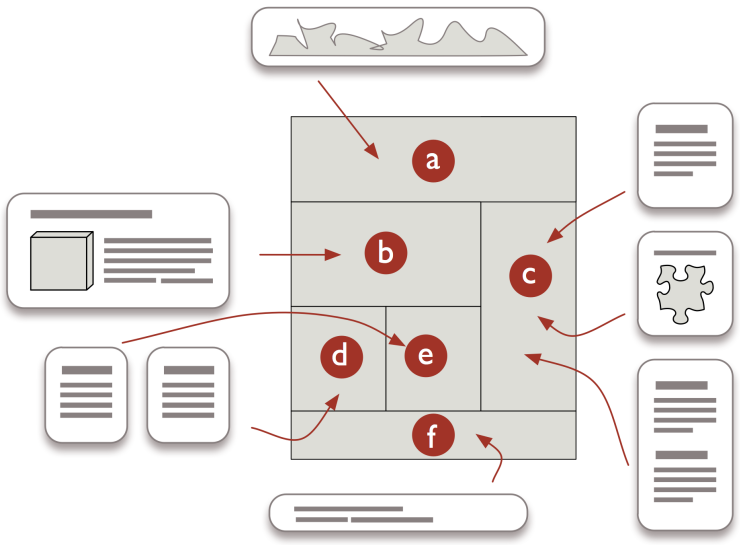


Figure 1. A layout made up of four rows and three columns that defines six slots (named *a*, *b*... *f*) where the contents of the element can be positioned.

```
body {
  display: "aaa" /auto
           "bbc" /225px
           "dec" /auto
           "fff" /auto
            * * 14em;
}
```

The dimensions of a template may depend on its content or be constrained setting certain values for the height of rows and the width of columns.

Finally, the contents of the template element can be positioned in any of the slots defined by the template, just indicating it with the `position` property, as for example:

```
#recent-entries  { position: d; }
#latest-comments { position: e; }
#about           { position: c; }
#blogroll        { position: c; }
```

Once the rationale behind the Template Layout Module has been introduced and a very simple use case has been shown, it is time to explain the syntax of the module in detail, which is done in following sections.

Template Definition

A **template** defines the layout of the element in which it is set, in terms of *rows* and *columns*. The proposed syntax for defining templates is as follows:

```
inline? [ <string> [ /<row-height> ]? ]+ <col-width>*
```

As it can be seen, instead of creating a new property, the module uses the `display` property, for which a new value is defined, consisting of one string of characters per each row in the template. Each string can be optionally followed by its height, and it is also possible to define a width for each column in the template. The actual syntax and behaviour for heights and widths is explained below.

An optional keyword `inline` may appear at the start of the value, indicating in that case that the element behaves as an inline-level element (templates can only be defined in block-level or floated elements).

Slots

As mentioned above, each string defines one row in the template, and each character in the string represents a column. These characters can be one of the following values:

a letter

Any alphabetic character represents a column in the template. Multiple identical letters in adjacent rows or columns are combined together to form a single *slot* for content, which spans those rows and columns.

`@` *('at' sign)*

It represents the default slot, that is, that in which will be placed the children elements for which no other slot has been specified.

`.` *('period' sign)*

A period creates an empty slot. No content can be positioned in it, and it is mainly intended to serve for defining *gutters* (empty columns that act as a separation among the actual columns of the grid that defines the template). But it can be used to create empty areas in the template or to separate any pair of slots, not just entire columns (or rows). This value would have not been strictly necessary, since the same effect could have been achieved with a normal slot (a letter) for which no content were specified, but it is provided as a convenience, because it helps reveal its intention, thus making the template more clear and easily readable.

Row Heights

A row definition may have associated a height. It can be an explicit length or other keywords that define certain constraints for the height of the row:

<length>

An explicit height for that row. It can be any valid CSS length. Negative values are not allowed and would make the template illegal (the declaration would be ignored).

`auto`

The height of the row is determined by its contents, according to the algorithm that is described later.

`*` *(asterisk)*

All rows with an asterisk have the same height.

Column Widths

Finally, it is also possible to define the width of each column. If column widths are present they must go after the last row in the template definition. They can be any of the following values:

<length>

An explicit width for that column. It can be any valid CSS length. Negative values are not allowed and would make the template illegal.

** (asterisk)*

All columns with an asterisk have the same width.

`max-content`

The width of the column is determined by its contents. This value means that the column tries to expand until fill the width required to fit its contents without any line break.

`min-content`

The width of the column is determined by its contents. In this case, the column will be as narrow as possible to fit its contents without it overflows.

`minmax(p, q)`

The width of the column is constrained to be greater than or equal to *p* and less than or equal to *q*, where *p* and *q* can be any of the following values:

```
[ <length> | max-content | min-content | * ]
```

If *q* is less than *p*, then *q* is ignored and `minmax(p, q)` is treated as `minmax(p, p)`.

`fit-content`

It is equivalent to `minmax(min-content, max-content)`.

Explanation of Width Values

The meaning of some of the previous values for the width of the rows is not clearly specified in the current version of the Working Draft, and it can only be understood deciphering the algorithm for computing widths (which is explained later). This is specially notorious for the values that let the width of a column be dependent on its contents: `min-content`, `max-content`, and `fit-content`. For this reason, their actual meaning is described below in some more detail.

A value of `min-content` for a column establish a constraint on that column under which it must be as narrow as possible while being able to display its content inside, without overflowing. In practice, this means that it will be as narrow as the wider word or image of its content. Let be the following example:

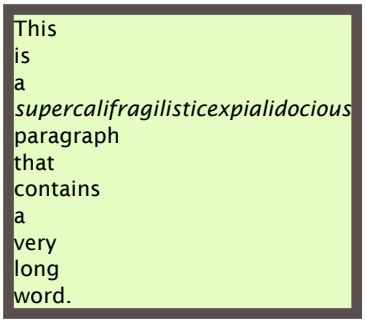
```
<p>This is a
<em>supercalifragilisticexpialidocious</em> paragraph
that contains a very long word.</p>
```

A column of a template with a `min-content` width that only contains a slot with that paragraph should behave as if the paragraph had been split into words (for example, inserting a `br` element after each word) and then floated or inserted in an HTML-table with just one cell (and `cellspacing` and `cellpadding` equal to zero). What most implementations do for such cases is shown in figure 2.

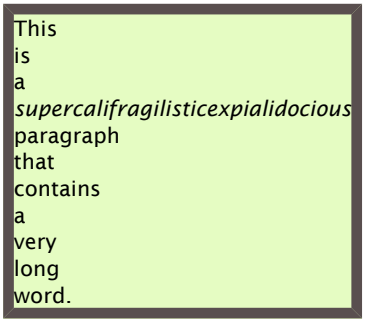
On the other hand, `max-content` means that the column should be as wide as the widest *line box* of the content of slots that span exactly that column (`rowspan=1`). Again, this can better understood taken as a reference what current CSS 2.1 implementations do for floats and tables. In this case, and in the absence of other constraints, that column would behave as would do a floated element or a single-cell table with the same table, as is shown in figure 3.

Note that if the content generates several line boxes, its maximum width remains unconstrained, and it tries to expand until fill all the available width.

Finally, if `fit-content` is applied to a column width, that column is constrained to be wider than the computed width for `min-content` and narrower than that for `max-content`.




```
<p style="float: left;">This<br/> is<br/> a<br/>
<em>supercalifragilisticexpialidocious</em><br/>
paragraph<br/> that<br/> contains<br/> a<br/>
very<br/> long<br/> word.<br/></p>
```




```
<table cellspacing="0" cellpadding="0">
  <tr>
    <td>
      <p>This<br/> is<br/> a<br/>
<em>supercalifragilisticexpialidocious</em><br/>
paragraph<br/> that<br/> contains<br/> a<br/>
very<br/> long<br/> word.<br/></p>
    </td>
  </tr>
</table>
```

Figure 2. Most implementations treat equal a floated element and a table of just one cell with the same content. This figure shows how a paragraph that have been split into words inserting a `br` after each word is rendered: *a)* when it is floated; and *b)* as a single-cell table. Except for a subtle difference on how borders are being rendered for the table, both boxes are exactly equal (they have the same dimensions).

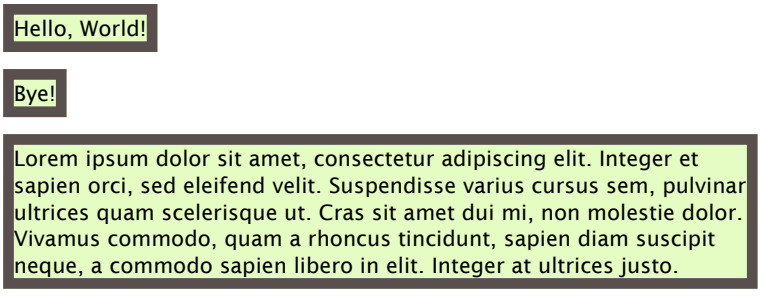


Figure 3. When a value of `max-content` is set for a column width, that row tries to expand horizontally until all its content can be displayed without line breaks. This behaviour is equivalent to that of floats or an HTML table with just one cell.

POSITIONING CONTENT INTO SLOTS

Actual content is assigned to specific slots in the template by means of the `position` property.

Once a template has been defined for an element, we can position the contents of that element in any slot of the template. This is done by means of the `position` property, already present in CSS 2.1, which now can also receive as a value a letter denoting the name of one of the slots in the template:

```
position: <letter> | '@' | same
```

Below is an explanation of the allowed values:

<letter>

The name of the slot into which the element is to be positioned. It must be one of the slots of its ancestor template. If no slot with that name is present, the computed value would be `static` (that is, it would behave as it were not positioned).

`@` *('at' sign)*

This usually will not be specified, since it is the default value for elements that are children of another with a template and for which no other slot have been specified. In that case, those elements go into the default slot of the template.

`same`

A value of `same` instead of a letter computes to the same letter as the most recent element with a letter as position that has the same template ancestor. If there is no such element, the value computes to `static`.

WIDTH ALGORITHM

The algorithm, as currently appears in the specification, is the following:

a) If the element where the template is defined has an a-priori known width:

- If the sum of the *intrinsic minimum widths* of the columns is larger than the width of the element, each column is set to its *intrinsic minimum width* and the contents will overflow.
- If the sum of the *intrinsic minimum widths* of the columns is less than or equal to the width of the element, the columns are widened until the total width is equal to the width of the element, as follows: all columns get the same width, except that no column or span of columns may be wider than its *intrinsic preferred width*. If the columns cannot be widened enough, the template is left aligned in the content area of the element (assuming a right to left direction language).

b) If the element does not have an a-priori width:

- If the sum of the *intrinsic minimum widths* of the columns is wider than the initial containing block, each column is set to its *intrinsic minimum width*. The resultant width of the element is the sum of the widths of the columns.
- If the sum of the *intrinsic minimum widths* of the columns is less than or equal to the width of the initial containing block, the columns are widened until the total width is equal to the initial containing block, as follows: all columns get the same width, except that no column or span of columns may be wider than its *maximum intrinsic width*. The resultant width of the element is the sum of the widths of the columns.

In addition, the rules for calculating **intrinsic minimum** and **intrinsic preferred** widths are defined as follows:

- A column with a `<col-width>` of a given <*length*> has intrinsic minimum and intrinsic preferred widths both equal to that <*length*>.
- A column with a `<col-width>` of * has an infinite intrinsic preferred width. Its intrinsic minimum width is 0.
- A column with a `<col-width>` of `min-content` has an intrinsic minimum width and intrinsic preferred width that are both equal to the largest of the intrinsic minimum widths of all the slots in that column:
  - The intrinsic minimum width of a `.` is 0.

  - The intrinsic minimum width of a letter or `@` is 0 if that slot spans two or more columns; otherwise, it is the intrinsic minimum width as defined by the CSS3 Basic Box Model (Bos, 2007b).
- A column with a `<col-width>` of `max-content` has an intrinsic minimum width and intrinsic preferred width that are both equal to the largest of the intrinsic preferred widths of all the slots in that column:
  - The intrinsic preferred width of a `.` is 0.
  - The intrinsic preferred width of a letter or `@` is the intrinsic preferred width as defined by the CSS3 Basic Box Model (Bos, 2007b).
- A column with a `<col-width>` of `minmax(p,q)` has an intrinsic minimum width equal to *p* and an intrinsic preferred width equal to *q*.

Explanation of Minimum and Preferred Intrinsic Widths

The algorithm above refers to some concepts intended to form part of the CSS3 Basic Box Model, but that have not yet been defined in the current version of that working draft (Bos, 2007b): the *intrinsic minimum* and *intrinsic preferred* widths.

They are closely related to the `min-content` and `max-content` values that have been defined before for <*column-width*> in this module.

Thus, the *intrinsic minimum* width of an element can be defined as the minimum width it must have to be able to display its contents without overflowing. In practice, this means that the generated box of that element will be as wide as the widest word or replaced element of its contents. Conversely, the *intrinsic preferred* width is the minimum width needed to display its contents without line breaking, if possible (that is, if that value is less than the available width for that element).

The above definitions for both values match those proposed by Baron (2007), which in turn correspond with the Mozilla extensions `-moz-max-content` and `-moz-min-content`[1]. The meaning of these width values is depicted on figure 4.

1 https://developer.mozilla.org/en/CSS/width

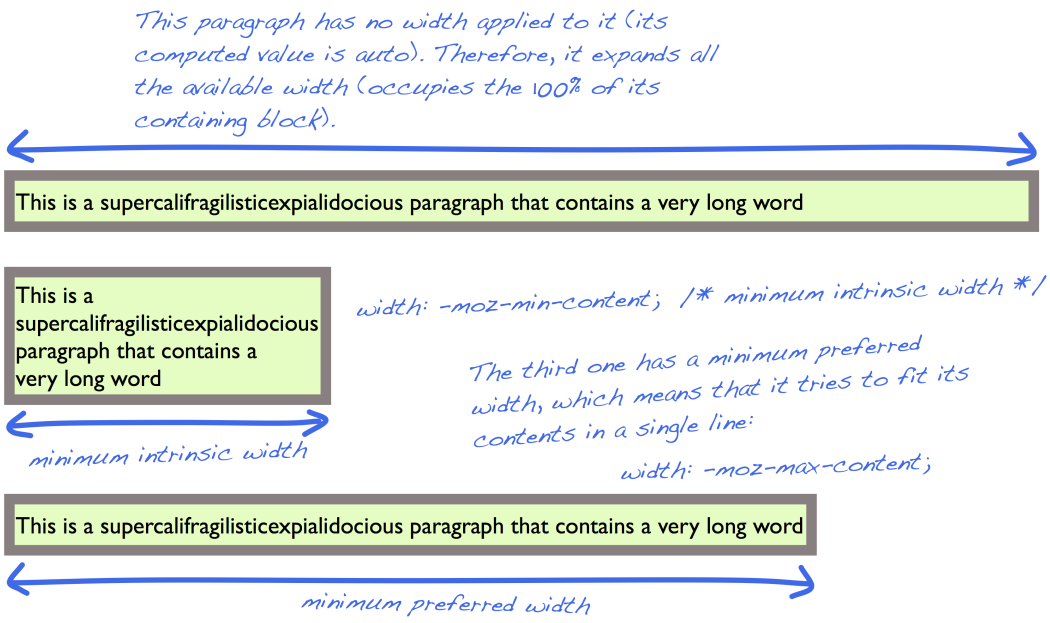


Figure 4. The same paragraph as is rendered in Mozilla Firefox when it has not any width set, with a *minimum intrinsic* width, and, finally, with a *minimum preferred* width (using the `-moz-min-content` and `-moz-max-content` Mozilla proprietary extensions of `width` property).

Height Algorithm

As it has been done for widths, first, it is described the algorithm for computing heights as it is currently described in the specification. Then, some clarifications are made, and certain omissions or errors in the specification are fixed. In addition, a much more detailed algorithm is presented here.

According to the specification, the height of the template is the smallest possible under the following constraints:

1 Rows with a height set to <*length*> have that height.
2 No row has a negative height.
3 All rows with a height set to * are the same height.
4 Every sequence of one or more rows of which at least one has a height set to `auto` is at least as high as every letter or `@` slot that spans exactly that sequence of rows.

5 If the computed value of the element's height is `auto`, then every sequence of one or more rows of which at least one has a height set to * is at least as high as every letter or `@` slot that spans exactly that sequence of rows.
6 The whole template is at least as high as the computed value of the element's height, unless that value is `auto`, or unless all rows have a height set to <*length*>.

As can be seen, the specification does not provide an actual algorithm for computing heights, but a set of constraints that any implementation of this module must fulfil. Although this is a common practice in W3C specifications, which usually leaves a great amount of freedom to implementors for choosing an actual algorithm, I think that this case is complex enough to deserve a more procedural description of *how* the constraints above can be fulfilled.

Moreover, there are certain omissions in the constraints above that, though obvious, should be explicitly set to avoid future misunderstandings in the implementations of the module. As Baron (2003) has stated (talking about the complexity of the layout engine of Mozilla): "I think some of the people who wrote the code didn't understand the specifications that they were implementing. Part of the problem may lie in the specifications themselves. For example, there's almost no information in CSS2 describing shrink wrapping."

For this reason, a concrete algorithm, written in a more procedural style, is provided below to help understand how all of those constraints can be achieved in practice.

Detailed Algorithm for Computing Heights

Computing Single-Row Slots

A first step of the algorithm for computing the height consists on calculating the minimum height of every row, considering only the slots of that row that do not span (`rowspan=1`). This step must be only done for rows for which a height value other than a explicit length have been set in the template definition. That is, only rows with a defined height of `auto` or * (*asterisk*) must be processed in this first step, since those with an explicit length already have their height constrained to that value.

Each row with a defined height of `auto` or `*` (*asterisk*) must be at least as tall as the tallest slot of that row that do not span (`rowspan=1`). The height of a slot, for the purpose of this algorithm, is determined by its contents. Of course, it depends on the width of the slot, so *computing heights must necessarily be done after computing widths*.

The pseudocode for this step is depicted below:

```
foreach row in template where row.height = 'auto' or
row.height = '*' do
  row.computedHeight = 0
  foreach slot in row where slot.rowspan = 1 do
    if (slot.getContentHeight() > row.computedHeight)
      row.computedHeight = slot.getContentHeight()
    end_if
  end_foreach
end_foreach
```

Rows of Equal Height

The Template Layout Module makes very easy to have equal-height rows, simply by assigning the selected rows a height of `*` (*asterisk*). The first step of the algorithm did not took this into consideration, and a minimum computed height was assigned to each row considering only the height of their slots. Now these rows must be traversed again, assigning to all of them a height equal to the largest of the minimum heights computed in the previous stage.

This is the second step of the algorithm:

```
var height = 0
foreach row in template where row.height = '*' do
  if (row.computedHeight > height)
    height = row.computedHeight
  end_if
end_foreach
foreach row in template where row.height = '*' do
  row.computedHeight = height
end_foreach
```

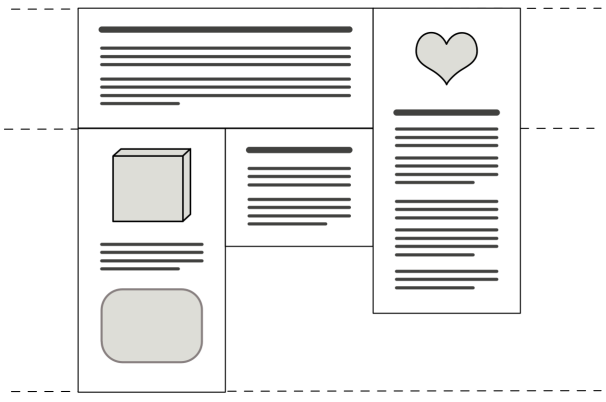


Figure 5. When a multi-row slot is shorter than the sum of the height of the rows it spans, it does not affect the overall algorithm for computing heights.

Computing Multi-Row Slots

The next step consists on computing the slots that spans several rows. Depending on the scenario, they may or not influence the computed height for rows calculated in the previous steps. Thus, let be the following template:

```
"aab" /auto
"cdb" /auto
```

And now let us suppose we have the scenario shown in figure 5 (where the slots have a height that only depends on their contents, since the algorithm for computing heights has not yet finished). In that figure the height of the slot `b` is less than the sum of the computed height of the rows it spans, which have been calculated in the two previous steps of the algorithm, so that slot is not affecting the computed height of those rows, and we will just have to "lengthen" it in a next stage of the algorithm until it is as tall as the sum of its spanned rows.

But, what does it happen when the contents of the slot that spans several rows are taller than the sum of the previously computed heights for the rows it spans? In that case, we need to increase the height of such rows —those that have a height of `auto` or *— until the sum of all of them is equal to the height of the slot with `rowspan > 1`. The specification does not stipulate how this must be done. Indeed, it even does not states that the the height of a slot that spans several rows must be equal to the sum of those rows. This is

something that should be stated in the specification, since it is clear that otherwise it could led to situations like the one shown in figure 5: it might seem obvious, but the specification should state that *slots must have the same height that the row to which they belong (or the sum of the rows they span)*.

Anyway, as to the case under discussion, that is, when the height of the spanned row is larger that the sum of the computed heights of the rows it spans, the lack of precision should not be considered an error of the specification: it is frequent in CSS specification that some concrete procedures are up to the implementors, who are free to choose different algorithms, as long as they fulfil the requirements for such specific parts of the specification. This is what happens, for example, with algorithms of tables.

In this case, two are the more obvious approaches to solve this issue while still preserving the constraints above:

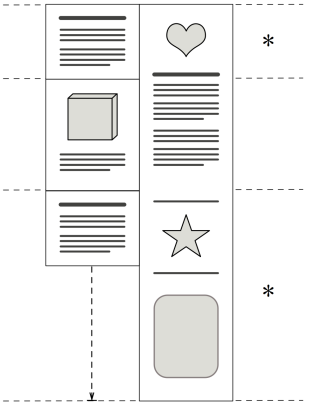
Distribute the excess of height among all the involved rows

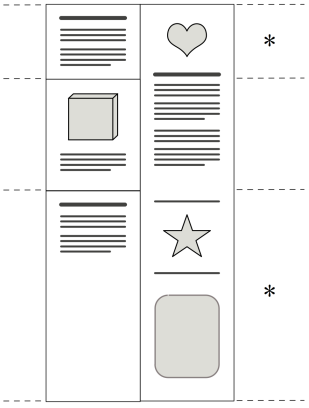
This would led to a more balanced design. The problem is: how should that length be distributed among those rows? One possibility would be to do it proportionally to their height. But, what height? The computed one? Or that specified in the template definition for that row? In the latter case, how should heights of `auto` and `*` be treated?

Simply expand one of the rows

A simpler approach could be just expanding one of the rows (for example, the last one), assigning to it all the difference between the height of the multi-row slot and the sum of the height of the spanned rows. This alternative is not without problems, though, because, what does it happen if such row is one of those with a defined height of `*`? If there are more equal-height rows among those which the slot spans, assigning all the excess of height to one of them would affect the others, which would see their height increased in the next step of the algorithm, and thus the multi-row slot should be enlarged to be as tall as the sum of the spanned rows. This could led, therefore, to situations like the one shown in figure 6.

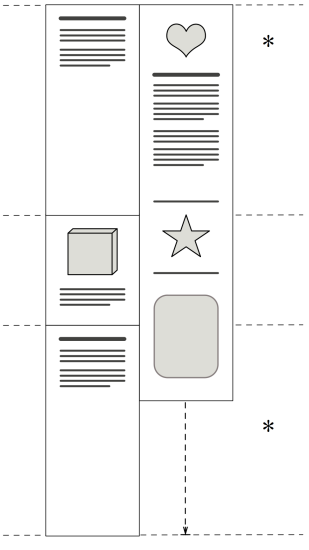
A multi-row slot (b) is taller than the sum of the height of the three rows it spans. Note that first and second rows have a defined height of * (*asterisk*), which means that they must have the same height, whatever their content.

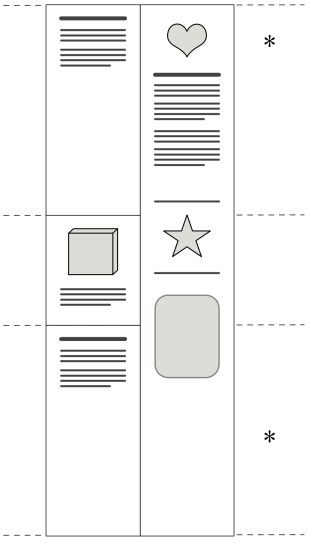




The second step of the algorithm for distributing the excess of height if an implementation decided just to enlarge the last row until the sum of all the involved rows is as tall as the slot that spans that number of rows. At this moment, it is violating the constraint of equal-height rows (those with a defined height of *).

A subsequent iteration should address the violation of the constraint on equal-height rows, ensuring that the rest of rows with a defined height of * are enlarged until all them are of the same height.





The final result, once assigned the same height to every row with asterisk. Although it fulfils all the constraints of the specification, the layout is less than ideal, since there are large unnecessary areas of whitespace.

Figure 6. The figure depicts the process of computing height for slots that span several rows, assuming that: *a)* the height of the slot is larger than the sum of computed heights of the rows it spans; *b)* an strategy consisting on assigning all the excess of height to the last row it is being used; and c) that row has a defined height of *, and there are other equal-height rows involved.

As it can be seen, assigning the excess of height (between the multi-row slot and the sum of the computed height of the rows it spans) to the last row is a less than ideal solution when that row is one with a defined height of asterisk and there are other equal-height rows among those over which the slot spans. In that scenario, we may end up with very large areas of unneeded vertical whitespace.

The *naive* alternative of simply assigning the excess of height to the last of the rows that a slot spans, though fulfils the constraints for computing height, may led to very unbalanced layouts with large areas of unnecessary whitespace, when there are equal-height involved. A much better alternative would be distributing the excess of height proportionally among all the spanned rows.

For that reason, and despite that solution fulfils all the constraints that have been set in this proposal, implementors are encouraged to choose an algorithm that distributes that excess of height *proportionally* among all the rows that the slot span. Since it is not possible to mix very different types of height that the specification allows (`auto`, `*` and an explicit length), it is necessary to *normalise* the heights first. The more obvious approach is to use the *computed height* of the rows, which has been calculated in the previous steps of the algorithm. The description of the algorithm would be, therefore, as follows:

First, it must distinguish between two cases:

1 If the height of the multi-row slot is less or equal than the sum of the computed height of the rows it spans
2 If the slot is taller than the sum of the computed height of the rows it spans

First, the height of all the rows that the slot span is normalised to the computed height that has been calculated in the preceding steps of the general algorithm. Then, the excess of height between the multi-row slot and the rows it spans is distributed proportionally to the computed value of each row. Note that, since equal-height rows —those that have a defined height of asterisk— have been already processed in the previous step, all of them are guaranteed to receive the same amount of length now, thus preserving the constraint that they must have the same height without the need of any special treatment for them.

To understand how the algorithm above works, let us consider a concrete example. Let be the following template, where the slot `b`

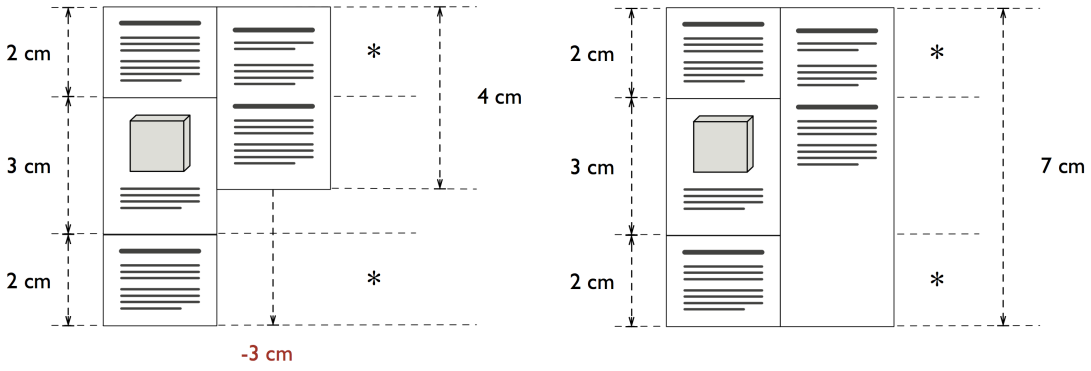


Figure 7. When the multi-row slot is shorter than the sum of the heights of the spanned rows computed in the previous steps of the algorithm, it does not affect the height of those rows, and this step of the algorithm just have to enlarge it until it has the same height than its spanned rows.

spans the three rows of the template, which have a height value of *, `auto`, and *, respectively:

```
display: "ab" /*
         "cb" /auto
         "db" /*
```

If the height of the contents of the slot `b` is smaller than the sum of the computed heights of the three rows that has been calculated so far in the preceding steps of the algorithm, it would not affect the height of its rows. All the algorithm has to do is enlarge the slot `b` until it is as tall as the sum of the height of all the rows it spans. Let us assume that the computed height of the rows is 2 cm, 3 cm, and 2 cm, respectively, and that the contents of the slot `b` are just 4 cm. That scenario is depicted in figure 7. The negative -3 cm means the difference between the height of the contents of the multi-row slot (`b`) and the sum of the computed height of the the rows it spans. All what must be done in this case is just enlarge the spanned slot until it has a height of 7 cm (2 + 3 + 2).

A slightly more complex scenario occurs when the height of the multi-row slot exceeds that of the rows it spans. In this case, the algorithm distributes that excess of height among the involved rows, proportionally to their computed height. This scenario is shown in

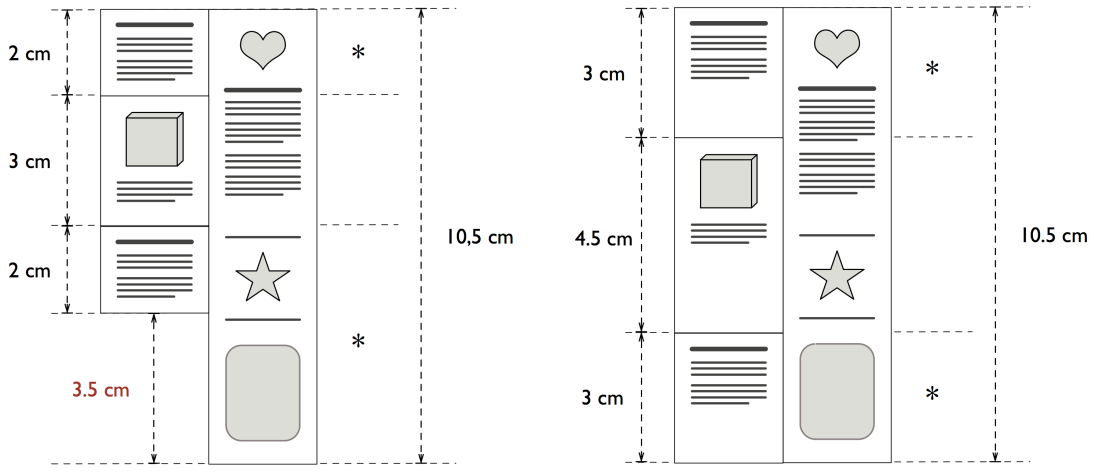


Figure 8. When the multi-row slot is larger than the rows it spans, the algorithm distribute that excess of height among all the spanned rows, proportionally to the computed height that has been calculated for each row in the previous steps of the algorithm.

figure 8, where the first and third row have a previously computed height of 2 cm (they have a defined height of *), and the second one has a height of 3 cm, given by the contents of the slot c. If the contents of the multi-row slot have a height of 10.5 cm, there is an excess of height of 3.5 cm, which must be distributed as follows:

- First and third rows receive an extra height of $2 \times 3.5 / 7 = 1$ cm
- Second row receives an extra height of $3 \times 3.5 / 7 = 1.5$ cm

As can be seen, the result is much more balanced than that obtained with the alternative of assigning the excess of height to the last row, which was shown in figure 6.

Processing again Equal-Height Rows

After computing the height of multi-row slots, it is necessary to repeat the third step of the overall algorithm, since in this process might have been affected rows with a defined height of *, as it was demonstrated in the last subsection. However, there is no need to perform any additional procedure, but just repeating the same step that was made before for rows of equal height.

Setting the Height of Slots

So far, we have computed the height of every row, but a final step must yet be done: it is necessary to assign to every single slot

(those that do not span more than one row) the height of its row. The algorithm is as simple as follows:

```
for each slot in template do
  if (slot.rowspan = 1)
    slot.height = slot.row.getComputedHeight()
  end_if
end_foreach
```

Overall Algorithm

To conclude this section, the overall algorithm for computing slots is shown:

1. Normalise the height of the rows
2. Compute the minimum height of each row
3. Compute equal height rows
4. Compute multi row slots
5. Compute equal height rows (*again*)
6. Set the height of each single row slot to fit that of its row

SLOT PSEUDOELEMENT

It has been shown how we can create templates and position elements into the slots they define. But, in order to achieve the pursued separation between presentation and content, there still is a use case that has not been addressed, namely, the ability to apply styles to the slot themselves.

A `::slot()` pseudo-element adds the possibility to refer specific slots from the style sheet to apply style to them.

Although missing in the first drafts of the specification, this feature was added to the version of 9 August 2007 (), where a pseudo-element `slot` was introduced to refer to single slots inside a style sheet. This is the syntax for such pseudo-element, as currently appears in the specification:

```
::slot(letter | @)
```

The `slot` pseudo-element selects the slot of the specified name. If the subject of the selector is not a template element, or if, being a template element, it has not such slot, the pseudo-element selects nothing (it still is a legal selector, though; it simply does not select anything).

This is an example of use of such pseudo-element, which sets a background colour to one slot of the following template and changes the vertical alignment of other two slots:

```
#content {
  display: "aab"
           "cdb";
}

#content::slot(b) { background: rgb(66, 52, 49); }
#content::slot(c),
#content::slot(d) { vertical-align: bottom; }
```

Currently, only a few CSS properties are allowed for the `slot` pseudo-element:

- All background properties (to set a background image or colour to the slot)
- The `vertical-align` property
- The `overflow` property

The question about why are these the only properties allowed for applying style to slots is discussed in the subsection of the same name on page 217.

Vertical Alignment

The `vertical-align` property of a `::slot()` pseudo-element can be used to align elements vertically in a slot. The effect is as if the hypothetical anonymous block that contains the contents of the slot is positioned as follows:

`bottom`

The content of the slot is aligned to the bottom: the bottom margin edge of the anonymous block coincides with the bottom of the slot.

`middle`

The content of the slot is vertically centered in the slot: the distance between the top margin edge of the anonymous block and the top of the slot is equal to the distance between the bottom margin edge of the anonymous block and the bottom of the slot. If the content overflows the slot, *it will overflow both at the top and at the bottom*.

`baseline`

The anonymous block that encloses the content is placed as high as possible under two constraints:

- The top margin edge of the anonymous block may not be higher than the top edge of the slot.
- The topmost baseline in the content may not be higher than the topmost baseline of content in any other slot in the same row that also has `vertical-align: baseline`. Baselines of content inside floats are not taken into account. Slots that span several rows are considered to occur in their topmost row.

For all other values, the content is top aligned: the top margin edge of the anonymous box coincides with the top edge of the slot.

DISCUSSION

Previous sections in this chapter have been written in a style as direct as has been possible, with little room for comments and opinions, since they were aimed to reflect the current state of the specification in a descriptive manner. Despite some parts have been discussed in more detail than they are in the specification, as for the algorithm for computing heights, they did not represent changes to the specification, but mere clarifications of certain aspects that might be confusing.

In the following subsections, though, the approach is different: once the syntax and behaviour of the Template Layout Module have been described, I will present some other, in my opinion desirable, features, that I have proposed but have not been yet accepted

by the Working Group to be included in the specification, explaining their rationale.

Alternative Syntaxes

The proposed syntax for defining the template has been somewhat controversial. No doubt, it may sounds strange the first time an author sees it, since there is no other property in CSS 2.1 that takes a similar value. However, the ASCII matrix provides, in our opinion, a simple manner of defining the template at a single place. Moreover, it is easy to understand the template as a whole just looking at the ASCII matrix: how many rows and columns it has, which slots it defines, as well as the rows and columns they spans, and the row heights and column width.

Of course, other syntaxes were considered while writing the first versions of the draft. Thus, another possibility would have been to introduce, for example, two new properties like `rows` and `columns` that take an integer as a value, indicating the number of rows and columns, respectively, of the template. But then it would be necessary to have a way to specify what slots the template contains, the rows and columns they span, the width and height of every column and row, respectively, etcetera. All of this information is now provided in a single property instead, which in addition is easily human-readable.

Default Widths and Heights

The default value of row heights has changed to `auto` instead of `*`.

Previous versions of the Working Draft set `*` (*asterisk*) as the default value for both row heights and column widths. This means that, by default, all rows would have the same height and all columns would have the same width. While for columns this might be an appropriate default width, I had clear that it would be much better if the default value for row heights were `auto`.

Thus, heights would always be tall enough to accommodate their content, which is the most common situation in any real web page. This has been corrected in the last version of the specification.

Using Percentages for Column Widths

At this moment, the Template Layout Module does not allow to use percentages for the width of columns. This is because it could led to inconsistencies if the sum of column widths expressed in percentages were greater than one hundred per cent. But HTML tables also suffer the same issues and it is yet possible to set the width of their cells in CSS with percentages (Bos et al. 2009, §17.5.2.2):

> *A percentage value for a column width is relative to the table width. If the table has* `width: auto`*, a percentage represents a constraint on the column's width, which a UA should try to satisfy. (Obviously, this is not always possible: if the column's width is 110%, the constraint cannot be satisfied.)*

Nothing prevent us from adopting the same behaviour for templates as browser vendors implement in their table-layout algorithms when percentages appear mixed with lengths, or if the sum of percentage widths exceed one hundred per cent.

Although the module is inspired in the more powerful mechanism of *grid systems*, most of which are based on equal-width columns, not all people who create web sites are graphic designers nor are aware of such design tool, while percentages are a very common value used in the layout of many web sites, specially for *liquid* designs (those that adapt their proportions to the width of the browser window). It is true that percentages can currently be simulated by the Template Layout Module. For example, a three-column layout of 20%, 50%, and 30%, respectively, would be as follows:

```
display: "aabbbbbccc";
```

But not every combination is possible (unless we use an enormous amount of columns to get widths like 12%, 30%, and 68%). And obtaining layouts that combine both proportional and fixed length columns is difficult and even more unpredictable than what we are trying to avoid by prohibiting percentages. For example, what is supposed to happen in the following case?

```
display: "aabbbbbccc"
         15em * 2em;
```

For all the above, I advocates for the inclusion of percentages as a legal value for being used in the Template Layout Module for column widths. My proposal for allowing percentages for column widths is as follows:

- If the template element has an explicit width (that is, other than `auto`, percentages are relative to that width)
- Otherwise, percentages are relative to the maximum available width, that is, the width of the container element

Styling the Slots Themselves

This was one of the first things I missed when the first version of our prototype, ALMcss, was being developed. In order to debug the layouts created with it, I tried to use a very rudimentary yet common technique for debugging CSS: adding a border to the slots generated by the tool —actually, absolute positioned divs— to reveal the underlying template (how many slots were generated, in what position, their actual dimensions, and so on). It was then when I realised that we had forgotten to include such ability in the specification: in that version of the Advanced Layout Module (Bos, 2005), there was not possible to select a slot from the style sheet to apply some style to it, and so did I tell it to Bert Bos in my first F2F meeting as a member of the W3C CSS Working Group, hold during the W3C 2006 Technical Plenary at Mandelieu-La-Napoule. I also proposed it later in a document sent to the Working Group mailing list (`w3c-css-wg@w3.org`) on 2 June 2006, where I argued why I though this feature was not only desirable, but essential in order to achieve a true separation between presentation and content.

Equal-height Columns

In that document, I stated that a simple layout like that of figure 9 could not be achieved with the Template Layout Module unless we could apply a background colour to the slot itself. As it has been shown in previous chapters, one of the major limitations of CSS is its inability to get equal-height columns, so this type of layouts, where columns have a background colour which extends all the way

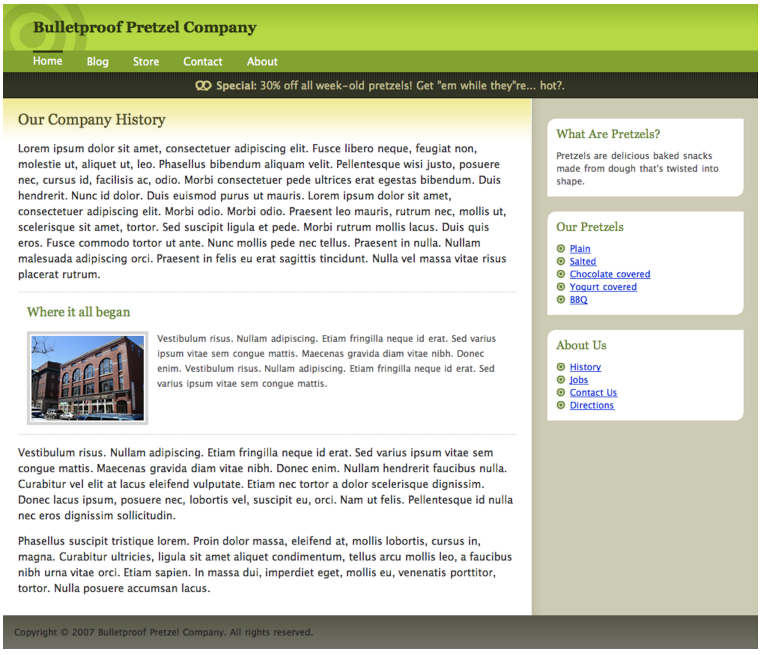


Figure 9. A requirement of the Template Layout Module is to get rid of those extra, non-structural divs that are added to the markup just for styling purposes, such as applying a background colour to an entire column, or setting some padding for the contents of a column without affecting the layout. Sometimes those divs already exist in the markup, but many others they have to be artificially added.

down the page, are usually made using the technique known as *Faux Columns* (Cederholm, 2004), using a background colour or a tiled image. In this example, Cederholm (2008, p. 258) is using an evolution of his own technique, known as *Sliding Faux columns* (Bowman, 2004, Meyer, 2004b), where the tiled image "can *slide* around behind fluid-width columns, thus creating the equal-height effect while remaining flexible" (Cederholm, 2008, p. 226). This is necessary because the columns of that design are 70% and 30% in width, respectively.

The basis of all this variations of the faux-columns technique consists of applying the background image or colour to the container of both columns.

But one of the major requirements of Template Layout Module is that it must avoid —or, at least, minimise— the need for such extra divs when they are not required by the structure of the content. Let us suppose that the rounded boxes that appear in the right column come from several places in the source code of the page. For

example, let us suppose that the markup for this design were that of figure 10.

In that case, the content of the right sidebar is not sequential and enclosed in a common container, but mixed in different positions with the rest of the content of the document. That layout, which is no longer possible in CSS2, could have been still done with the Template Layout Module using a code like that shown in the right column of the same figure. Since we have removed all the non-structural containers from the original markup, we need to be able to specify a background colour or image to the slot `c` (the sidebar).

My proposal was accepted and the pseudo-element `slot()` that has been described previously in this chapter (see p.) was introduced in the following version of the working draft (Bos, 2007a, §3.10). So for the background colour of the sidebar in the previous example we could have just done:

```
body::slot(c) {
  background-color: #D5D6BE;
}
```

What Properties Must Be Allowed?

Nevertheless, at this moment there is no consensus in the Working Group about what CSS properties must be allowed inside `slot()` pseudo-elements. In my opinion, limiting them to just background properties, `vertical-align`, and `overflow` is too restrictive.

For example, another common problem of current multi-column layouts with CSS is setting the horizontal separation (*gutters*) among columns when they are expressed in different units than those of the columns. Thus, the example used in this section sets the width of both columns in percentages, to 70% and 30%, respectively. What does it happen if we want a horizontal separation between columns of, for instance, 1 em, or 20 pixels? This is an issue in CSS2, which does not allow to use expressions for width values, as for example: `width: 70% - 1em`. In a case like this we have the following options (Cederholm, 2008, p. 218):

- Use a percentage for padding as well, and subtract that value from the declared width of the column.

Structure of the content (HTML)	Template-based layout (CSS)

<body>
   Bulletproof Pretzel Company
  <ul id="menu">...
 	
   What Are Pretzels?
  ...
 	
   Our Company History
  ...
   Where it all began
  ...
 	
   Our Pretzels
    ...
 	
   About Us
    ...
 	
</body>
	body {
	  display: "aaaaaaaaa" /auto
	  "bbbbbbccc" /auto
	  "ddddddddd" /auto;
	}
	
	h1, #menu {
	  position: a;
	}
	
	#history {
	  position: b;
	}
	
	#what, #store, #about {
	  position: c;
	}

Figure 10. A possible content structure for the Bulletproof Pretzel Company example.

- Apply padding to elements inside the columns only.
- Add an additional div to separately assign padding using any value we wish.

The second method consists of applying the padding individually to direct children of the columns:

```
#content > *, #sidebar > * {
  padding-left: 1em;
```

```
  padding-right: 1em;
}
```

But it is usually not so easy, either, specially with `em` values, because unless every children has the same font size the computed values of their paddings will be different, and it would be necessary to calculate the right `em` value for each element based on its font size.

Since that calculations might be cumbersome, in practice is very common the third solution of those enumerated above: adding an extra div inside each column:

```
<div id="sidebar">
  <div>
     ...
  </div>
</div>
```

Now, any padding value can be applied to that inner div without affecting the width of the column:

```
#sidebar {
        float: right;
        width: 30%;
}
#sidebar > div {
        padding-left: 1em;
  padding-right: 1em;
}
```

But this sort of things are those that we are trying to avoid with the Template Layout Module. In our example, this could have been achieved as follows:

```
body {
  display: "aaaaaaaaa" /auto
           "bbbbbbccc" /auto
           "ddddddddd" /auto;
}
```

```
body::slot(c) {
  padding-left: 1em;
  padding-right: 1em;
}
```

Note that the template definition could be easier if, as I stated in the previous subsection, percentages for column widths were allowed:

```
body {
  display: "aa" /auto
           "bc" /auto
           "dd" /auto
           70% 30%;
}
```

But currently this is not possible using the Template Layout Module since it does not allow the use of `padding` properties with the `slot()` pseudo-element. It can be argued that this is because, as it has been stated in this chapter, it is more oriented toward *grid systems*, which are based on equal-width columns with a fixed separation (*gutter*) among them. In a grid-based layout, therefore, the horizontal separation among columns is supposed to be achieved using the empty slot ("."). Thus, the following template would led to the layout shown on figure 11.

```
body {
  width: 900px;
  display: ".a.a.a." /auto
           ".b.b.c." /auto
           ".d.e.c." /auto
           1em * 1em * 1em * 1em;
}
```

But, as I have already stated, for most users I think that the use of `padding` is more intuitive. Moreover, strictly speaking, what we get by using empty columns as *gutters* are more close to margins than to paddings. And, although only horizontal spacing has been considered so far, there also is the same issue with vertical spacing.

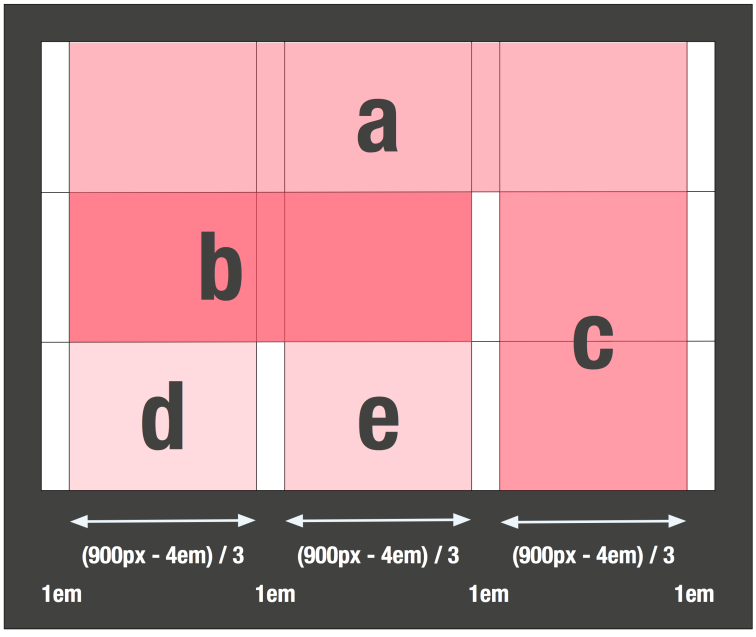


Figure 11. Template Layout Module allows mixing proportional columns widths with fixed separations among them (*gutters*), something typical in *grid systems* in which it is inspired. Nevertheless, I still think that `padding` and other properties should be allowed for `slot()` pseudo-elements, since it is what authors are used to. I do not see any reason why we should treat slots different in that sense than, for instance, table cells, which can be styled normally in CSS2.

And there are other properties that I would like to be allowed too. For example, authors usually specify the font family and size based on the placement of elements. Thus, the font size of a sidebar, for instance, is often smaller than that of the main content, different colours are sometimes applies to its headers, etcetera. I do not see any reason why these properties should be prohibited in the `slot()` pseudo-element.

As a conclusion, my proposal is allow most CSS properties inside the `slot()` pseudo-element, with the only possible exceptions of:

- `float`
- `margin` and `margin-` related properties

Non-rectangular Slots

Currently, the Template Layout Module does not allow non-rectangular slots for content. This is a restriction inherited from the cur-

rent CSS box model, which prevent non-rectangular boxes. But we should at least consider if such constraint must be still valid.

For example, by removing it, it would be very easy to get a design like that shown on *a previous chapter* (see figure 10 on page 147) and obtain something similar to what is possible with floats, but where is the box itself (the slot) and not just its contents, what *flow* around another slot, thus getting a "L"-shape:

```
display: "ab"
         "bb";
```

I proposed to remove this constraint in the CSS-WG F2F meeting at CWI (Amsterdam, May 9-11, 2006), where I illustrated it with the example cited above. Then, an interesting debate followed, in which several scenarios of non-rectangular layouts were depicted on the whiteboard. Following there is a brief extract of the minutes that reflects that discussion, though very condensed by the scriber ("C" it is me):

> C: *"demo. 3 column layout"*
>
> MM: *"Meta issue: What is the flow model in advanced layout (can you flow content from one box into another). Can this be non square (like an L-shape layout)"*
>
> SZ: *"Quick proposal: Define incursions anchored on 4 corners."*
>
> SZ: *"in a 9 grid scenario you can have an object in the middle and flow around…"*

Figures 12 and 13 show the two scenarios depicted by Steve Zilles which are mentioned in the minutes above.

The problem of removing this constraint and allowing non-rectangular slots, as was shown by several members of the Working Group in that meeting, is that it can led to layouts where it would be difficult to standardise how the content must flow. That is the case, for instance, of the layout shown in figure 13. Which direction must follow the content of the slot `a`? From left to right? From top to bot-

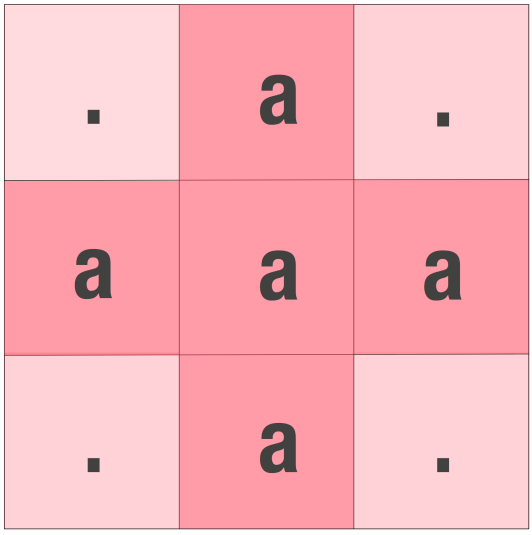


Figure 12. An example of non-rectangular slot (*a*), sketched by Steve Zilles at the F2F meeting of the CSS Working Group at CWI (Amsterdam, May 9-11, 2006), during the interesting discussion which followed to my proposal of removing from the Template Layout Module the constraint that forbids non-rectangular slots. The figure shows incursions (in this case, empty spaces, but they could have been normal slots) anchored on the four corners of the layout.

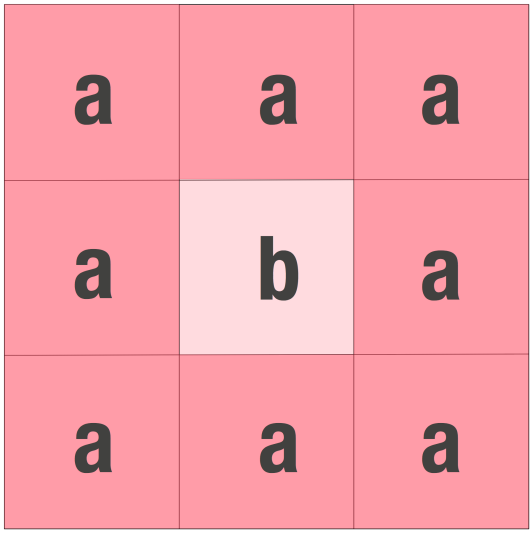


Figure 13. The second example is also from Steve Zilles, during the same meeting. In this case, a more complex scenario is shown, in which there is a slot (*b*) in the middle of the 3 × 3 grid, and the rest of the content (slot *a*) flows both horizontally and vertically around it.

tom? Both? In my opinion, this should not be an insurmountable problem, as long as CSS3 also plans to introduce a multi-column module, of which this could be thought a particular case. But I agree that these scenarios may be out of the scope of the Template Layout Module, at least in this initial stage of its existence, and it is probably better not to make it very complex at this moment, both for authors and for implementors.

However, I still think that this feature could be interesting for getting simpler layouts that are not, however, currently possible with CSS2, like the one that opened this subsection, so I propose the following intermediate approach: to remove the constraint that prevent non-rectangular slots, replacing it by this other constraint: *the characters (letters, @, or .) of a slot must be adjacent*.

10

Demonstration: Case Studies Revisited

This chapter is aimed to demonstrate the benefits of the proposed solution, and how it gives a step forward towards a true separation between presentation and content on the web, and, more specifically, between the structure of the document and its final visual layout.

The approach followed for such demonstration consists on taking the case studies that have been done on Chapter 7 using the current layout capabilities of CSS and redoing them, but this time using the template-based advanced layout mechanism proposed on Chapter 9.

INTRODUCTION

One of the benefits of the CSS3 Template Layout Module is, as this chapter will reveal, how easy it is to specify the layout of a page or a specific element with it, specially when compared with how the same layout would have been done with the current CSS 2.1 mechanisms reviewed in chapters 5 and 6, to wit: floats, negative margins, and relative and absolute positioning. To proof this statement, the case studies that were done on *Chapter 7* with CSS will be revisited, now laying them out using the aforementioned template-based layout mechanism proposed in this thesis.

But, whereas the simplicity is one of the major advantages of the Template Layout Module, where it really stands out is when it comes to redesign. Therefore, this chapter also undertakes a complete redesign of a real website, first briefly explaining how it could have done in CSS —actually, why it is *not* possible to be done without altering the order of the content and the structure of the markup— and then illustrating how it would be with the proposed layout extension.

BLOG ENTRIES

This section will show how with the same HTML structure for a blog entry that was outlined in p. 133 (and is more detailed below), it is possible to do any of the layouts that were seen in the blogs reviewed on *Chapter 7*. This is the HTML that I am going to use for all the examples (although some new elements that are not contemplated here may appear in some of the examples).

```
<div class="entry">
  <h1>Lorem Ipsum</h1> <!-- Title of the post -->
  <ul class="meta">
    <li class="date">
      11 <abbr title="December">Dec</abbr> 2009
    </li>
    <li class="categories"> ... </li>
```

```
      <li class="tags"> ... </li>
      <li class="num-comments"> ... </li>
      ...
    </ul>
    <div class="content">
      ...
    </div>
    <div class="comments">
      ...
    </div>
  </div>
```

Let us suppose that we want the date of the post to appear above the title of that entry. As we know, this is only possible in CSS using absolute positioning and leaving room for the the date with either a `padding-top` for the `#entry div` or a `margin-top` for the `h1`, depending on each specific case. This may lead to the already mentioned problems of absolute positioning, which can be summarised saying that as soon as the font size or the browser window dimensions change, the absolute positioned date might overlap the content below it.

Instead, this could have been solved with the Template Layout Module, using a template like the following for blog entries:

```
.entry {
  display: "a"
           "@";
}
```

And then indicating the position for the date with just this single property:

```
.entry .date { position: a; }
```

Every other information contained in the blog entry (title, other metadata, comments, the content itself, etcetera) would be placed below it, in the default slot defined by '@'. Since the height of the rows is not defined, it will be equal to its default value, `auto`, and

therefore the rows will automatically adapt to the height of their contents, whatever they be, under any circumstance.

Despite for this example a very simple template like the above suffices, nothing keep us from doing more complex variations over the content order. In fact, the design of any of the blogs reviewed on *Chapter 7* could have been achieved without changing the previously outlined HTML for blog entries.

Zeldman

Thus, for Jeffrey Zeldman's blog (see p. 134), a template like this could have been defined:

```
#entry {
  display: "a"
           "@"
           "b"
           "c";
}

.entry .date       { position: a; }
.entry .categories { position: b; }
.entry .comments   { position: c; }
```

Stuffandnonsense

The entries of the Andy Clarke's blog (see p. 135) at his design agency would be laid out as:

```
.entry {
  display: "@@"
           "aa"
           "bb"
           "cc"
           "de"
           61% 39%;
}

.meta              { position: a; }
#promo-workshop    { position: b; }
```

```
.section h2         { position: c; }
.comments           { position: d; }
.archive            { position: e; }
```

Unless other slot is specified, all the entry content goes into the default slot, located at the top of the page. That includes the title (`h1`). The metadata of the entry (for which Clarke is using an `entry-meta` class and here it is represented by `meta`) would go below the entry, regardless of its position in the source document. Below, it is the announcement of a workshop. Comments and "From the archives" sections are situated at the bottom of the page.

Finally, as it can be seen in figure 2 of page 135, the main content of the blog entry is narrower than the information that is below it. Specifically, in Clarke's original web site it is set a width of 55%. This could have been done adding two more empty columns at both sides of "`@`" and "`a`" slots:

```
.entry {
  display: ".@."
           ".a."
           "bbb"
           "ccc"
           /* What should be here? */
           * 55% *;
}
```

But there is a problem in the template above: how should be defined the row for the entry comments and the archive section? A simple solution like `"dde"` would not work, because Clarke has defined those sections as follows:

```
/* This is the class of the "replies" section */
.article {
  position: relative;
  float: left;
  width: 57%;
  padding-right: 4%;
```

```
    border-right: 2px solid #e6e6e6;
}
```

That gives as a result that the comments section has a width of 61% (actually, 61% plus 2 pixels corresponding to its right border), and therefore the adjacent archive section will have 39%. But those values are in conflict with the previously defined width of 55% for the central column (distributing proportionally the rest between the left and right columns).

There are two possible solutions for this (none of them are currently allowed in the Template Layout Module Working Draft, though):

Using a nested template

A nested template could have been defined for comments and archive. But, although this feature is allowed in the current working draft, there is no guarantee that we always have an appropriate HTML element where the nested template can be defined, which would break again the separation between structure and layout (if a new element had to be added just for acting as a container of the two sections).

This would have been easily solved, though, if we remove the constraint that limit the properties that can be applied to `slot` pseudo-elements. By doing so, the following code could have been used for this example:

```
.entry {
  display: ".@."
           ".a."
           "bbb"
           "ccc"
           "ddd"
           * 55% *;
}

.entry::slot(d) {
  template: "ef"
```

```
            61% 39%;
}
```

Using paddings

The other option would be, using the first suggested template, simply to define a padding for the narrower parts of the entry (again, the same mentioned constraint should be removed from the specification for this to be possible):

```
.entry::slot(@), .entry::slot(a) {
  padding-left: 22.5%;
  padding-right: 22.5%;
}
```

Meyerweb

As for Eric Meyer's entries (see p. 136), the following template would do the work:

```
.entry {
  display: ".a"
           "bc"
           "dd"
           10em max-content;
}

.entry h1         { position: a; }
.entry .meta      { position: b; }
.entry .content   { position: c; }
.entry .comments  { position: d; }
```

Mark Boulton

The layout of Mark Boulton's entries (see p. 137) could have been easily done with the following simple template:

```
.entry {
  display: "a"
           "@";
}
```

```
.date { position: a; }
```

Stopdesign

Douglas Bownman's posts (see p. 138) have the following layout:

```
.entry {
  display: "aa"
           "dc"
           14em max-content;
}

.entry h1       { position: a; }
.entry .meta    { position: d; }
.entry .content { position: c; }
```

Jason Santa Maria

The Jason Santa Maria's blog entry showed in 139) is the most complex of the reviewed ones. However, it is very easy to recreate its layout with the Template Layout Module without any changes to my original HTML for a blog post:

```
.entry {
  display: "aaaaaaaaa"
           ".bcccccde"
           "ffgggggggg"
           ".hiiiiijk"
           "mmmmmmmmm"
           90px;
}

.entry h1           { position: a; }
.entry .num-comments { position: b; }
.entry .categories   { position: c; }
.entry .prev         { position: d; }
.entry .next         { position: e; }
.entry .tags         { position: i; }
.entry .comments     { position: m; }
```

News

Similarly to blog entries, news articles in a newspaper may have a plethora of different layouts: sometimes the summary of the news article is below the headline, others over it; some articles have an associate photography and others have not, and, for those with a photography, this can adopt infinitude of sizes and positions within the article. However, as it happened with blog posts, it is clear that all news articles should have the same HTML, regardless of the layout of each single piece of news.

Let us use the following HTML for representing a news article:

```
<div class="news">
  <h2>This is the Headline</h2>
  <p class="summary">
    ... <!-- The summary goes here -->
  </p>
  <div class="main-picture">
    <img src="..." alt="..."/>
    <p class="caption"> ... </p>
  </div>
  <div class="content">
    ... <!-- Content goes here -->
  </div>
</div">
```

How could be laid out, for instance, the news that appears on the *National Post* page shown in 143? Although any change in the position of the headline, summary, and main picture is very easy to achieve with the Template Layout Module, those news constitute a good example that, in its current form, it is not yet a solution for some layouts, and that my proposal of allowing non-rectangular slots has sense.

Thus, assuming that non-rectangular slots were possible, the layout of the main news article of that page could have been done as follows:

```
#news1 {
  display: "aaaaaa"
           "bbbbbb"
           "cddddc"
           "cccccc"
           "cceecc";
}

#news1 .headline          { position: b; }
#news1 .summary           { position: b; }
#news1 .main-picture      { position: d; }
#news1 .content           { position: c; }
#news1 .content .picture  { position: e; }

/* This property does not form part of the Template
   Layout Module: it is defined in the CSS3
   Multi-column Module. */
#news1 .content {
  column-count: 6;
}
```

The news article "American museun…" of the same page would use the following template:

```
#news2 {
  display: "aaaa"
           "bccc"
           "cccc"
           "ccdd";
}

#news2 .headline          { position: a; }
#news2 .summary           { position: b; }
#news2 .main-picture      { position: d; }
#news2 .content           { position: c; }

#news2 .content { column-count: 4; }
```

Master in Web Engineering

The website of the Master in Web Engineering at University of Oviedo that was shown in the figure of 146 is also an example of how non-rectangular slots could serve for certain layouts, as it was explained during the review of such case study.

However, I will ignore here such discussion and concentrate only on doing the whole layout of that website using the Template Layout Module; it would be as follows:

```
body {
  display: "aaaaaa"
           "bbbbbb"
           ".ccc.d"
           ".e.f.d"
            15px 240px 15px max-content 15px
185px;
}
```

The template above would be enough for defining the layout of that web site. It is not only much less code than it is needed using floats or absolute positioning, but the really important thing is that it is totally independent on the order of the document source code. Content may be represented in the markup attending only to their logical structure, and they could be later positioned in any of the defined slots, no matter where they have been defined in the HTML.

BIOTinfo Magazine

Again, I am tackling a problem that would require non-rectangular slots to be accomplished. I am referring to the page of BIOT local magazine that was shown in 150, and, more specifically, to the article that appears on the bottom and that was studied in pages 149–153.

```
body {
  display: "aaa"
```

```
            "abb";
}

#animations { position: a; }
#biot-centenaires {
  position: b;
  display: "@c"
           "@@";
}
#biot-centenaires img {
  position: c;
}
```

STYLING A DEFINITION LIST

This is one of the case studies analysed on previous chapter (see p. 156) where the advantages of the Template Layout Module over current CSS properties is more evident. In that example, it was shown how it was needed to combine floats with negative margins to put each list element (`dt` and `dd`) into its desired position. Even without changing the order in which they appeared in the HTML document, it was not an easy task. Moreover, that layout is using a fixed height for `dt` elements (otherwise, it would have been impossible to arrange the subsequent elements).

Although the design was explained on *Chapter 7* in detail (pp. 153–163), it will be summarised here to make easier the comparison between what must be done in CSS 2.1 and how it would be with the proposed template layout mechanism:

- *All elements are floated to the left* (see figure 1). An explicit width (in pixels or other unit or percentages, but known a priori) and height must be necessarily used for `dt` elements, since those are the distances that `dd` elements have to be moved downwards and to the left in the following step.
- *"dd" elements must be moved downwards an to the right*, a distance equal to the height and width, respectively, of the `dt` elements. In this case, relative positioning is not an option, since the boxed must be

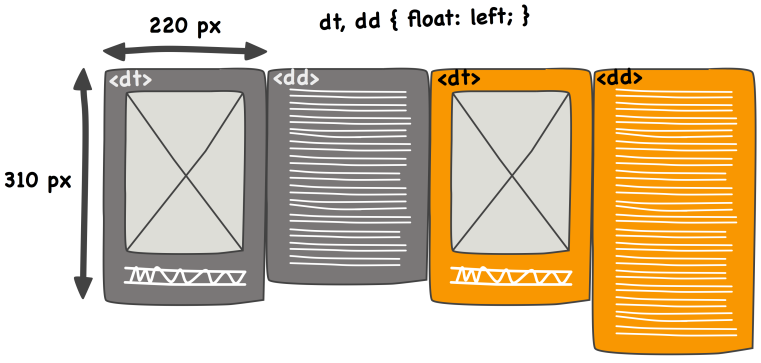


Figure 1. The first step in the Caxigalines case study consisted on floating every `dt` and `dd` elements to the left. Using current CSS properties the desired final layout is only possible if a fixed width and height is established for `dt` elements, since those will be the same values to be applied to the margins of `dd` elements for moving them both vertically (downwards) and horizontally (to the left).

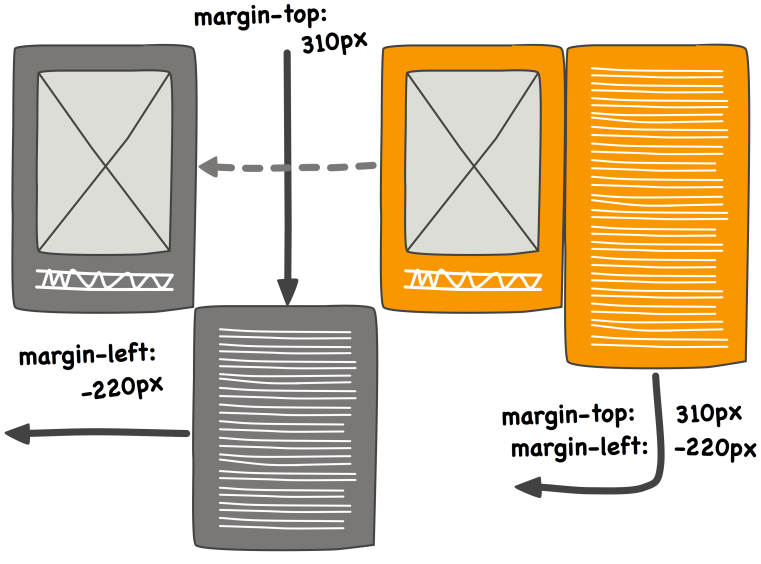


Figure 2. Top and left margins are applied to `dd` elements to push them down and to the left, placing them below the previous `dt` element. Note that the top and left margins have to be equal to the height and width of the `dt`, which must therefore be known a priori (they can not adapt to fit their contents). Margins, unlike relative positioning, actually moves the element. Therefore, by moving `dd` elements we also make room for the following `dt` element on its right to put itself on the empty space left by the `dd`.

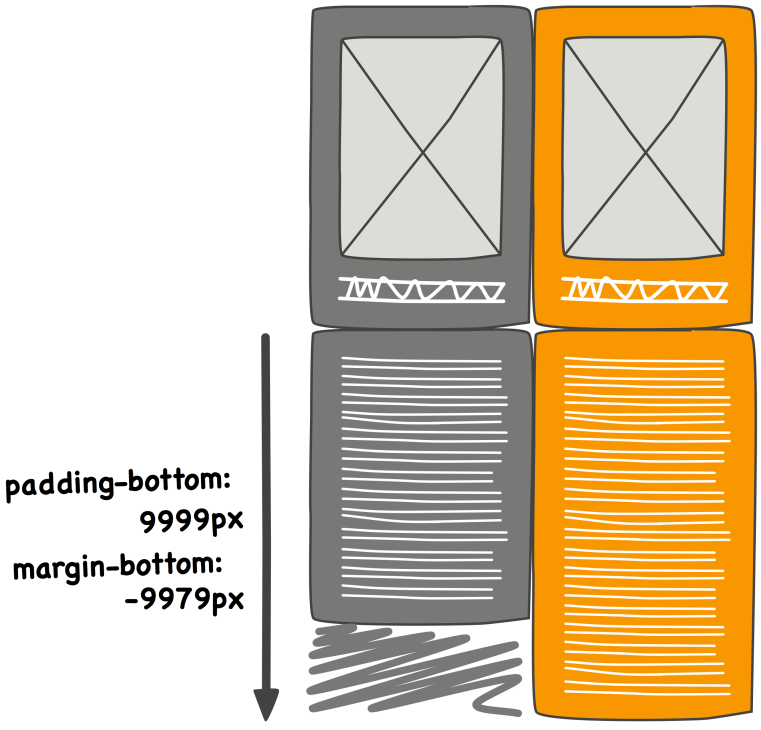


Figure 3. This is the final appearance of the definition list, once their elements have been floated to the left and every `dd` have been moved below its preceding `dt`. Although still a last step is needed to assure that every element have the same height. Either the faux columns technique (assuming fixed widths and only two rows) or the approach followed by One True Layout can be applied.

actually moved from its current position, and margins (including negative ones) must be used to put each `dd` underneath its preceding `dt` (figure 2).

- By pushing down the `dd` elements, the following floated element in the source document (that is, the following `dt`), has empty room and, according to float rules, is automatically moved to the left until it touches the right edge of the preceding floated `dt`. The final result is shown in the sketch of figure 3.

Once we have summarised the layout process with the current layout capabilities of CSS, let us see how the same layout could have been achieved with the Template Layout Module:

```
#categories {
  display: "abc"
           "def";
}
```

```
.alimentacion { position: a; }
.hogar        { position: b; }
.moda         { position: c; }
.papeleria    { position: d; }
.tecnologia   { position: e; }
.otros        { position: f; }
```

That is all. And, actually, it is not the *same* layout. Note that no heights nor widths are specified in the template. That means that the layout would automatically adapt to any change in the font size, browser window dimensions, type of font, or the contents of each element. Not only the layout is much easier to achieve, but it also resists any change in any of the mentioned factors, which are not under the control of the designer.

The layout thus done is much easier, bulletproof, and order-independent than with current CSS mechanisms. Moreover, a completely different layout can be achieved using the Template Layout Module in a matter of seconds, instead of hours with floats and margins —assuming it were even possible—, by simply defining a new template (a change in a single property, without doing any calculation).

But, even more important, the design so built is totally independent on the content order. Thus, changing the position of the list items would be so simple as to interchange the letters in the `position` property.

Finally, a completely different layout can be done almost instantly, with no more than changing the template (that is, the value of the `display` property). For instance:

```
#categories {
  display: "aac"
           "aad"
           "bbf"
           "edf";
}
```

YoDona Magazine

The more complex of the layouts reviewed in the case studies chapter is one of the better examples of the improvement that the Template Layout Module would mean for creating CSS layouts. In this case, it is an unordered list (see an excerpt of the markup on 169) that must be laid out following a complex 4 × 5 grid (see figure

28 on 167), where each module of the grid have different dimensions and orientation (figure 29, p. 168).

Notwithstanding its complexity, the layout can be achieved with the Template Layout Module with a few properties. First, the template is defined:

```
#agenda {
  display: "abcde"
           "fghij"
           "kllim"
           "noopp";
}
```

Although the above is not the only possible template, for this example I consider that it is what most simplify the layout process. As it can be seen, instead of defining a single slot for each module, I am using two slots: one for the image and another for the description of the event. By doing so, I am able to do all the layout with the Template Layout Module, without recurring to absolute positioning and paddings inside each slot for arranging the images. In addition, by doing so it is also possible to simulate non-rectangular slots, as it is required for the case of `#juliettelewis`, where the image spans two rows (slot `i`), whereas the description is contained in a single cell (slot `h`).

```
#diesirae            { position: b; }
#diesirae img        { position: a; }
#lacintablanca       { position: d; }
#lacintablanca img   { position: c; }
#marabu              { position: j; }
#marabu img          { position: e; }
#almudenabaeza       { position: f; }
#almudenabaeza img   { position: g; }
#juliettelewis       { position: h; }
#juliettelewis img   { position: i; }
#loscondenados       { position: l; }
#loscondenados img   { position: k; }
#maisonmumm          { position: n; }
```

```
#maisonmumm img      { position: o; }
#muchomas            { position: p; }
#muchomas img        { position: m; }
```

Widths and heights can be explicitly set for columns and heights or, as in the template above, they can be left unspecified, thus obtaining a fully liquid layout (both horizontally and vertically) that *nevertheless retains the vertical alignment without overlapping*.

As for the last example (*Styling a Definition List*), the main benefit of using the Template Layout Module is appreciated when the layout needs to be changed. Thus, a completely different layout could have done, for instance, as follows:

```
#agenda {
  display: "aaakll" /150px
           "bbbkll" /40px
           "bbbggi" /300px
           "bbbffh" /250px
           "oonncd" /250px
           "ooeecm" /125px
           "jjeepp" /170px
            180px 180px 120px 150px 180px 180px;
}
```

The result of applying the template above can be seen in figure 4, as shown using the ALMcss prototype presented in this thesis.
A more traditional layout, using a single column, could have been achieved with a template like this (the result, as it is rendered in a real browser by ALMcss is shown in figure 4):

```
#agenda {
  display: "baa" /200px
           "cdd" /200px
           "jje" /225px
           "gff" /250px
           "ihh" /250px
           "llk" /200px
```

Figure 4. An alternative, completely different layout for the agenda page of YoDona magazine, made with the Template Layout Module, simply changing the template.

AGENDA

[CINE, MÚSICA, LIBROS, ARTE, ESPECTÁCULOS, DANZA, FOTOGRAFÍA, TEATRO, SOLIDARIDAD]

DIES IRAE
Marta Carrasco
viernes
13
La bailarina y coreógrafa fusiona danza y teatro en esta revisión furiosa e irreverente del Réquiem de Mozart para 15 intérpretes. Hasta el 22 de noviembre en el Festival de Otoño de Madrid. **MÁS INF.:** TEATROABADIA.COM

LA CINTA BLANCA
Haneke en Segovia
miércoles
18
La última película del director austriaco inaugura la IV Muestra de Cine Europeo de la ciudad castellano-leonesa (Muces). El certamen homenajea al director Jaime Chávarri y tiene a Reino Unido como país invitado. **MÁS INF.:** MUCES.ES

LA BARCELONA CANALLA
Màrius Carol
"
El periodista y escritor echa la vista al siglo XX para desvelar, en *Las plumas del marabú*, la intrahistoria erótica y sentimental de una Barcelona capital inesperada de la *dolce vita* europea. Publica La Esfera de los Libros. **MÁS INF.:** ESFERALIBROS.COM

ALMUDENA BAEZA
Somos/Nos gustaría ser
*
La artista nos muestra esta instalación sobre la distancia entre la realidad y el deseo en el espacio de arte Frágil (Madrid). La pieza se extiende en forma de caálogo (a la dcha.), en edición limitada **MÁS INF.:** FRAGILESESPACIODEARTE.BLOGSPOT.ES

JULIETTE LEWIS
Salvaje rockstar
miércoles
25
Con la carrera como cantante viento en popa, Lewis vuelve a sudar y desmelenarse con su aquelarre de rock primitivo y arañazos punk. En la Sala Heineken (Madrid). **MÁS INF. Y FECHAS:** TICKETMASTER.ES

LOS CONDENADOS
Bárbara Lennie
viernes
20
La actriz demuestra sus dotes dramáticas en la última cinta de Isaki Lacuesta (*Cravan vs. Cravan*), una denuncia de los traumas dejados por las dictaduras en Latinoamérica, en relación con los desaparecidos y su memoria. **MÁS INF.:** BARTONFILMS.ES

MAISON MUMM
Cultura en Burbujas
lunes
16
La bodega GH Mumm reproduce en el hotel AC Santo Mauro (Madrid) el *château* de origen en Reims, para desplegar toda su sabiduría en torno al mundo del champán, con una exposición y tienda, ediciones exlusivas, catas, diálogos con sumilleres... Hasta el 21 de noviembre. **MÁS INF.:** AC-HOTELS.COM

Más, mucho más
JOSÉ CARLOS PLAZA
Bodas de sangre
El director subraya la fatalidad y la sexualidad de esta pieza fundamental en la obra de Lorca. Hasta el 3 de enero en el Teatro María Guerrero (Madrid). **MÁS INF.:** CDN.MCU.ES

Figure 5. **Another, more traditional layout of YoDona magazine. As for the example before, it has been achieved just changing the template.**

```
            "oon" /300px
            "ppp" /170px 180px 100px 180px; }
```

ONE TRUE LAYOUT

This section will redo using the Template Layout Module not a case study of *Chapter 7* —all of them have already been reviewed in this chapter—, but the One True Layout technique that was studied on *Chapter 6*. As it was described there (p. 122), this technique achieves any number of columns, *in any order*, where all columns are also the same height, using current CSS capabilities. This section will review how the same effect can be obtained much more easily with template layout.

I will use for this demonstration the same example that I made for explaining that technique (see figures 12 and 15).

Using the Template Layout Module, a 3-1-2 layout like the one that was explained in that chapter would be as easy as:

```
#content {
  display: "312";
}

#orange     { position: 1; }
#strawberry { position: 2; }
#lime       { position: 3; }
```

With just those properties we are achieving the desired order and equal-height columns, using a liquid layout. But, unlike One True Layout, we can obtain hybrid layouts that mix fixed-width width liquid columns, using as many different length units as we want. For example, the following template would create a three-column layout where the first two columns have a width of 120 pixels and 18 em, while the third one is liquid:

```
#content {
  display: "312";
           120px 18em max-content;
}
```

Figure 6. Any variation of the original One True Layout can be achieved just with a simple change of letters in the template, even if it involves several rows, or different number of columns. The figure shows how the template below is rendered in a real browser using ALMcss, the prototype developed for this thesis (one of them).

Finally, the major strength of template layout comes when more drastic changes are required. Thus, it is possible to get the layout of figure 6 with only using this template:

```
#content {
  display: "332"
           "111";
}
```

Figure 7. The old version of Stopdesign web site that is going to serve as a starting point for my demonstration of how a complete redesign of a real web site can be made with the Template Layout Module without changing the markup, which, in addition, has been cleaned and is even more structural than the original one.

A COMPLETE REDESIGN

The last section of this demonstration chapter will be devoted to, first, recreate the layout of a real web site using the Template Layout Module, and, then, tackle a complete redesign of such layout so that it looks like a completely different real site. All of that with a purely structural markup. The web site I will use for this example is an old version of Stopdesign[1], the consultancy firm of the renowned designer Douglas Bowman (see figure 7).

1 www.stopdesign.com

```
<body>
  <div id="header">...</div>
  <ul id="nav">
    <li id="current">Home</li>
    <li><a ...>Portfolio</a></li>
    ...
  </ul>
  <div id="highlights">
    <h2>About Stopdesign</h2>
    <p>A design consultancy ... </p>
    ...
    <h2>From the Portfolio</h2>
    ...
  </div>
  <div id="latest">
    <div class="entry">
      <h2><a href="/log/2006/05/27/
going-to-google.html"
      >Going to Google</a></h2>
      <p>The cat&#8217;s out of the
bag....</p>
      ...
    </div>
  </div>
  <div class="entry">
    <h2><a href="/log/2006/05/26/
webstocking.html"
    >Webstocking in NZ</a></h2>
    ...
  </div>
  <div class="entry">
    <h2><a href="/log/2006/04/29/
google-calendar-tips.html"
    >Google Calendar tips</a></h2>
    ...
  </div>
  <div class="previously">
    <h2>Previous Entries</h2>
    <dl>
      <dt><a href="/log/2006/04/13/
google-calendar.html"
      >Google's new dating
game</a></dt>
      <dd>...</dd>
      ...
    </dl>
  </div>
  <div id="latestlinks">
    <h2>Latest Links</h2>
    <ul class="offsite">
      ...
    </ul>
  </div>
  <div id="frequentedsites"></div>
  <div id="organizations"></div>
  <div id="feeds"></div>
  <ul id="about">
    <li>About</li>
    <li>Events<</li>
    ...
  </ul>
  <div id="footer">...</div>
</body>
```

Figure 8. A sketch of what could be a purely structural markup for www.stopdesign.com.

This is somewhat subjective, of course, since I am not the author of that site. But, for the purposes of this example, let us suppose that this is the most logical order of the content. How could then the layout shown in figure figure 7 be made?

First, given that the navigation menu appears at the top of the page and it is below it in the source code, the only choice that we have is to use absolute positioning. And the same can be said of the other navigation bar which appears near the end of the HTML and must be laid out just below the header. As it has repeatedly said in this dissertation, this has the drawback that the height of the navigation bars and the header must be explicitly defined so that they can be placed one below the other without overlapping.

Next, the sidebar with the highlighted content can be placed on the right by using `float: right;` and assigning it an explicit width.

Finally, what happens with the two columns that appear below the latest entry? It could also be easily done by floating them to right. But then we need add an extra `div` to wrap the elements that make up the middle column. And, again, this requires them to have an explicit width.

So I have been able to do the layout with a few CSS2 properties and almost no changes in the original HTML. Where is the problem then?

As always, the first problem with which we encounter in CSS is the **complexity**. The explanation above has been oversimplified for brevity, but it is not as easy as it might seem, as it has been thoroughly explained in this and previous chapters. Let us compare the actual design with how it would have been accomplished using template layout (see figure 9)

Changing the Layout

But let us go further and think what would happen if we wanted a completely different layout. For example, one like that of A List Apart[1]. Figure 10 shows the final appearance of Stopdesign redesigned, for the purposes of this example, to look "like" A List Apart.

1 www.alistapart.com

```
#container
{
  display-model: "a"
                 "b"
                 "c"
                 "d"
                 "e";
}

#nav      { position: a; }
#header   { position: b; }
#cnav     { position: c; }
#footer   { position: e; }

#content
{
  position: d;
  display: "oop"
           "qrp";
}

.entry          { position: r; }
#lastest .entry { position: o; }
#highlighted    { position: p; }
#links          { position: q; }
#previously     { position: r; }
```

Figure 9. The Stopdesign home page as it could be laid out with the Template Layout Module.

Note that I am completely breaking the linear order of the elements in the markup: all the elements in the `highlights` section are now distributed across several columns and mixed with elements coming from other sections in the original document; some elements from the `footer` are now in the right column, as well as the

Figure 10. Stopdesign home page with the layout of A List Apart

`company-navigation`; the most recent article now goes to the end of the middle column; etcetera.

Everybody will agree that this would be impossible today with CSS without many changes in the HTML source code. Nevertheless, with the Template Layout Module it could have been easily achieved (stylistic changes aside) with the following template:

```
body
{
```

```
  display-model: "aaaaa"
                 ".bcd."
                 "eeeee"
                 225px * 230px 230px 155px;
}
```

And then putting the pieces of the document into the desired slots:

```
#nav, #header          { position: a; }

.entry,
#previously,
#about                 { position: b; }

#portfolio,
#mostrecentarticle,
#events                { position: c; }

#searchform,
#latestlinks,
#frequentedsites,
#organizations,
#host,
#powered,
#feeds,
#cnav                { position: d; }

#footer              { position: e; }
```

Conclusions

As it has been shown with this case study, even if only for the *simplicity* that it brings to the design process, the Template Layout Module would be a major improvement over current CSS layout mechanisms, where it really stands out is when it comes to **redesign**. This is a consequence of the much more independency between the structure of the content and the visual layout that it achieves with respect to floats, without the drawbacks of absolute positioning.

11

ALMcss: A JavaScript Implementation of Template Layout Module

This chapter presents ALMcss, the first available implementation of the CSS3 Template Layout Module. It is a JavaScript prototype that, once included by a web page, allows to specify its layout using the new properties and values defined in that module, and works on most current web browsers.

Of course, if the solution presented here is eventually accepted to become a W3C Specification, are browser vendors who should implement it natively. But, in the meantime, notwithstanding it is merely a prototype and, as such, has some bugs and does not implement all the features of the proposed solution, it has served as a proof of concept that such proposal is not only feasible, but it can be in fact implemented using just JavaScript and the DOM.

Acknowledgements

The development of ALMcss, the prototype that implements in current browsers the solution presented in this thesis, was initially funded by a research project of CTIC Foundation, and then subsequently refactored and improved in several students' master theses.

The initial version of the prototype developed for this thesis was funded by CTIC Foundation[1] (Centre for the Development of Information and Communication Technologies in Asturias), through a research project with Fundación Universidad de Oviedo[2] (FUO), named *Extensión del estándar CSS3 que permita la adaptación multidispositivo de contenidos web* (project code FUO-EM-115-05, code name ALMcss). It was the award-winner research project in the *I Premios Sociedad Información en Asturias* (First Asturias Information Society Awards), promoted by the regional government of the Principality of Asturias.

The project, of a duration of one year, consisted on a research grant that was first held by María Rodríguez, who thus became the developer of the first implementation of the Template Layout Module[3]. When she got another position, after doing an invaluable job, her grant was occupied by Miguel García, who refactored the code and solved some bugs. It is compulsory to express my gratitude to both of them, as well as to CTIC Foundation, which graciously allowed me to donate the project to the research and web communities, making it publicly available under a W3C License.

I can not forget, either, Enrique Cabal, a student of mine who, for his B.S. in Software Engineering undergraduate thesis developed a layout engine, both in Java and C#, that implemented the Template Layout Module with a subset of HTML and CSS, and who, for his master thesis in Computer Science, continued the previous work of María and Miguel, refactoring the prototype and improving the layout algorithms ().

1 http://ctic.es

2 http://www.funiovi.org

3 Strictly speaking, of the Advanced Layout Module, as it was formerly named (Bos, 2005). It has not been until the public Working Draft of 2009, April 9 (Bos, 2009) when its name changed to Template Layout Module.

Introduction

A short history of the development of ALMcss has already been told in the first chapter of this dissertation, and therefore it will not be repeated here. Instead, this chapter focuses on describing the design of the prototype. As it has been noted in the summary of the chapter, ALMcss (acronym for *Advanced Layout Module for CSS*) has been the first and, for more than three years, only available implementation of the CSS3 Template Layout Module presented here as the solution proposed by this author to the problem of layout on the web, and was developed while it was a "W3C Members Only" Working Draft. It was first presented in *World Wide Web Conference* (Bos & Acebal, 2006) and then in Clarke's book (2007a), to which I was honored to contribute (Clarke & Acebal, 2007).

Having a prototype that implements, to a certain extent, the Template Layout Module, not only has allowed me to demonstrate that it should not be difficult to implement natively by browser vendors (which is, of course, the ultimate goal if it is eventually adopted as a W3C Recommendation to form part of the CSS3 specification), but, since I made it publicly available trough a W3C License, and despite its many bugs, web designers has also been able to *see* the possibilities of the template-based positioning in practice, working in a real browser, instead of having to figure out how it works just reading the specification.

It has not been until very recently when a second implementation of the module has appeared, also as a JavaScript prototype (Deveria, 2009). Today, more than three years after ALMcss was developed, there is not any major browser that supports it, even experimentally.

Before describing the design of the prototype, the following section discusses the alternatives considered for its development, and a second prototype that was later developed for this thesis, following another of such approaches, is also briefly introduced.

State of the Art

The first phase of the research project consisted on a study of the alternatives for the implementation of the prototype. The four different approaches that were analysed are enumerated below, and then explained in some more detail in next subsections:

Changing the code of an open source browser

Since there are quite a few good browsers, including some industrial ones, that either are open source or are based on an open source layout engine, the first obvious alternative would be to take one of them and add the new layout functionality to it. This would have the benefit of allowing us to concentrate on the new layout features, since the rest of the CSS implementation should be left to the browser. As it will be seen, this is not so in practice.

Creating a layout engine from the scratch

Another option would be to develop a prototype from the scratch. Of course, given the time constraints (one year, and just one developer), we could not pretend to implement a fully compliant CSS 2.1 browser that in addition supported the more advanced layout features of Template Layout Module. But it would be possible to develop a minimal HTML/CSS parser and layout engine that did not care of most CSS properties, such as those of fonts, list styles, borders, etcetera, and concentrate only on the new layout capabilities.

Implementing an extension of an existing browser

Some browsers let developers extend their functionality through *extensions*, also known as *add-ons*, being Mozilla Firefox the one that counts with the most impressive number of them, including some well-known ones by web developers, such as Firebug[1] or Web Developer Toolbar[2]. It seems to be, thence, a reasonable alternative to be considered for the development of ALMcss.

1 http://getfirebug.com/
2 http://chrispederick.com/work/web-developer/help/

Developing a JavaScript plugin

A slightly different version of the latest alternative was to develop not an extension for a concrete browser, but a JavaScript prototype that every web page could include and, once loaded by the web browser, accessed to the underlying HTML document and CSS rules through the Document Object Model (DOM) to retrieve the new layout properties and then modified the document accordingly, after computing the position of the so positioned elements.

Changing the Code of an Open Source Browser

This was the first and probably most obvious alternative considered for the development of the prototype. Given the standard compliant browsers that either are open source, like Mozilla Firefox, or are based on some open source CSS layout engine, as is the case with Apple Safari and the open source engine WebKit, they seemed good candidates to be modified to include the new layout functionality. Therefore, a significative amount of time was devoted to study their source code and developer documentation to analyse the viability of this alternative.

Although the results of such study are too extent as to be detailed in this dissertation, the main conclusions of our review will be outlined. To do this, I will rely on the study of one specific open source browser: Mozilla Firefox, which main architecture is depicted in figure 1. Some comments about the open source layout engine WebKit will also be made.

Mozilla Structure of Subdirectories

A web browser, and, specifically, its layout engine, is a complex piece of code. Adding to it new properties or, in this case, new values for existing properties (those defined by the Template Layout Module for the `display` and `position` properties), and the behaviour associated to these new values, would require changes in many components of the browser, from the CSS parser to the layout engine. Although this is not the place to describe them in detail, some of the most important directories in the code source structure of Firefox are the following (Mozilla, 2009a):

- **browser**: contains the main classes of the browser itself.

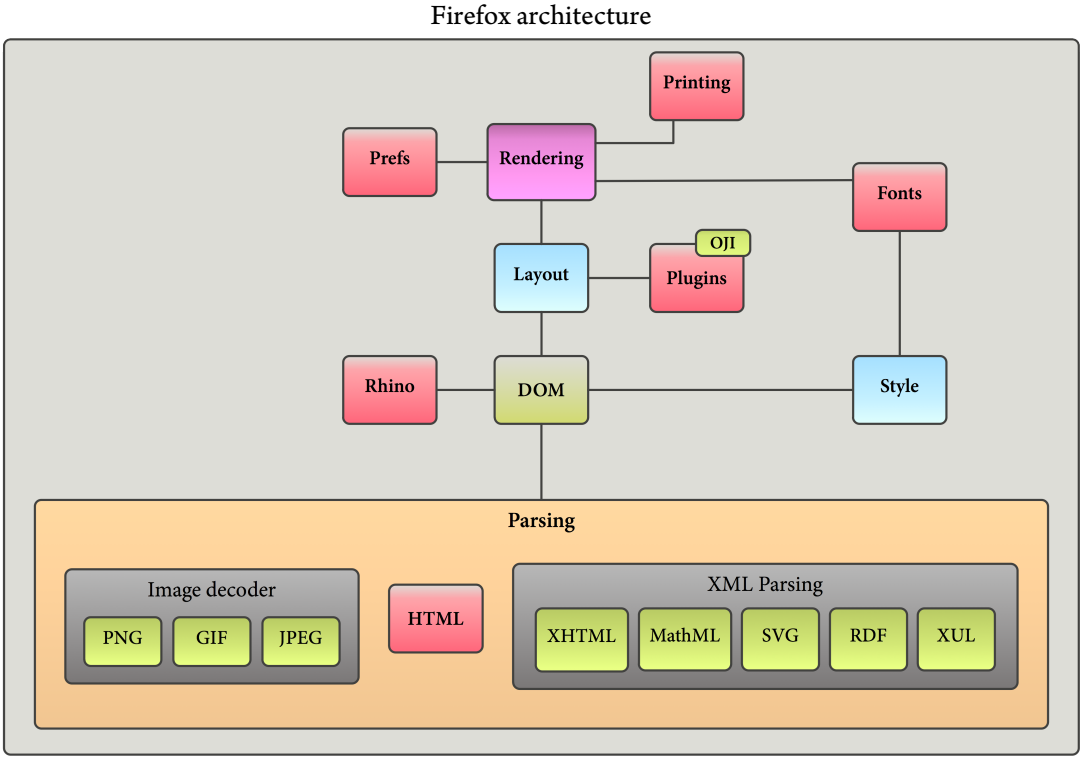


Figure 1. Firefox architecture.

- **content**: was taken apart from *layout* and contains objects that are exposed to the DOM.
- **dom**: contains C interfaces and code for implementing and tracking DOM objects in Javascript. It forms the C substructure which creates, destroys and manipulates built-in and user-defined objects according to the Javascript script. For example, if the Javascript script adds a user-defined attribute to the document (such us `document.goofy = 1`), this code will create the `goofy` node, put it on the `document` node and manipulate it according to any later Javascript commands.
- **layout**: it is the core of the layout engine, which decides how to divide the real state of the window among all the pieces of content. It is the responsible of resizing and arranging the content accord-

ingly to CSS1, CSS2, alignment styles and the content itself. It *does not display the content*, but just computes its position and size (which is known as *reflowing* the document). This code is also known as `NGLayout` and **Gecko**.

- **parser**: contains the HTML and XML parsers.

Adding a New Style Property

Summarising the process described in Mozilla (2009c), adding a new style property would require to modify at least the following parts of the source code:

- **CSS property names and hint tables**: the system must be formally informed of the existence of the new property name, modifying the following classes and interfaces: `nsCSSPropList.h`, `nsChangeHint`, `nsStyleConsts.h`, and `nsCSSProps.cpp`.
- **CSS declaration**: the declaration must be able to hold the new property and its values. To do this, changes are needed to the structs and classes defined in `nsCSSDeclaration.h` and `nsCSSDeclaration.cpp`.
- **CSS parser**: the parser must be able to parse the property name, validate the values, and provide a declaration for the property and value
- **Style context**: the `StyleContext` must be able to hold the resolved value of the property, and provide a means to retrieve the property value. Additionally, it has to know what kind of impact a change to this property causes.
- **Rule nodes**: the `RuleNodes` need to know how the property is inherited and how it is shared by other elements.
- **DOM**: the style should be accessible from the DOM so that it can be dynamically retrieved or modified.
- **Layout**: layout has to know what to do with the property, that is, the meaning of the property, its actual behaviour, and how it affects to the rest of the layout process.

Too Complex and Poorly Designed Code

The problem is not to have to make changes in all the modules excerpted above, since some of these changes are, after all, unavoidable (it is clear, for example, that adding a new property will necessarily imply to modify the parser code), so much as the fact that the overall design of Firefox and, specifically, its layout engine, *Gecko*,

```
// Whether or not we can collapse our own margins with our children. We don't
// do this if we had any border/padding (obviously), if we're the root or HTML
// elements, or if we're positioned, floating, a table cell.
m_canCollapseWithChildren = !block->isRenderView() && !block->isRoot()
        && !block->isPositioned() && !block->isFloating()
        && !block->isTableCell() && !block->hasOverflowClip()
        && !block->isInlineBlockOrInlineTable();
```

Figure 2. A code excerpt from `RenderBlock.cpp` class in WebKit source code. Only that class is about five thousand lines of code long, and it is plenty of duplicated conditional logic like the outlined in this figure. And it is just one of the many C++ classes that are involved in the layout process. Certainly, it is not the better example of good software design that one can find. In addition, it is almost undocumented (although it is true that, at least, significative names are used for variables and methods).

leaves much to be desired in terms of good object orientation and use of design patterns. And the same happens at an implementation level, with a C++ code that many times resembles more a "C with classes". Some reasons for the complexity of the layout engine have been suggested by Mozilla's engineer Baron (2003), and they include the great number of people involved, and an obsession for excessive optimisation in some areas (and, when this happens, is always the code readability that suffers).

Other of the layout engines compared in our review is WebKit (2010), on top of which Apple Safari (and now Google Chrome) are built. It has been traditionally accredited as having a cleaner code than Gecko, and this is probably true with regards to the coding style. But, when one looks at its design, essentially the same pitfalls can be found. Just as an example, let us consider the WebKit class responsible for laying out block boxes, `RenderBlock`. It has 5080 lines of code. That is not an error per se, since it is true that layout, as it has already been stated, is a complex task. But it is full of conditional logic like the outlined in figure 2 that could have been avoided if some design patterns had been applied.

Why do not simply use a single condition like this?:

```
if (block->canCollapseMarginsWithChildren()) {...}
```

Lack of Documentation

Another major problem which both Gecko and WebKit suffer is the lack of documentation. Being so complex pieces of code, one would expect to find an extent documentation where their overall architecture and detailed design were explained: which classes are responsible of what, class diagrams explaining its static structure, sequence diagrams that show the interaction in execution time between objects involved in complex process like layout and rendering, written explanations of what design decisions have been taken and why, etcetera. Nevertheless, such sort of documentation is nonexistent in both cases, or, at least, it is very difficult to find. It is true that digging into the developer mailing lists, or browsing the internet, some information can be found, in form of presentations (Baron, 2008; Fisher, 2009; Baron, 2006a), blog posts, etcetera. But it is not the type of documentation that would be expected in so big software projects.

Conclusions

Implementing the Template Layout Module in one of the major open source browsers would have been, no doubt, a great achievement for this thesis. But it also turned out to be an even more complex task than it seemed to be when we began our review. Not only for the complexity of code itself —something that is an essential feature of a layout engine that implements a complex specification like is CSS 2.1—, but for the following two reasons that have already been pointed out in the paragraphs above:

- The lack of a proper documentation
- A design that is not so object-oriented as it could be

The former issue would not be insurmountable, had it not been for the latter. Despite the absence of an appropriate design documentation would certainly be an inconvenience —specially when dealing with large-scale code of more than one hundred thousand lines of code, as is the case of both Gecko and WebKit—, it could have been worked out if the overall design were understandable. But we found out that, although the WebKit code seems to be simpler than that of its counterpart, Gecko, in both cases adding a new feature (even a single CSS, as it has been shown), requires modifying many parts of the code. Specially in our case, where a completely new positioning

scheme had to be implemented, it would imply to modify the core of the layout engine, which behaviour is scattered over many classes, something that exceeded the time and resource constraints of our research.

In addition, although seeing the Template Layout Module implemented in a *real* browser would have been very rewarding, it did not have, either, the same benefits, in terms of public visibility, as others of the considered options. It would have been served perfectly well to the purpose of proving that the Template Layout Module could be implemented, of course, but… how many people would be willing to download a modified version of a layout engine and experiment with it? (unless our changes were later incorporated to the real browsers that use it, of course, in which case the situation would be the opposite and the achieved visibility would be the maximum).

For all the above, this option was discarded.

Creating a Layout Engine from the Scratch

Of course, if the alternative of modifying an existing browser was discarded for its complexity, by creating a layout engine from the scratch we do not mean to implement a fully CSS 2.1 compliant browser, something that would have been unfeasible in a short amount of time. But what does not seem unrealistic is to develop a prototype, using a high-level object-oriented programming language like C++, Java, or C#, that implements only an (X)HTML parser and a minimal subset of CSS (for instance, colour related and other easily implementable properties, leaving aside the complexity of margins, floats, absolute positioning, tables and the rest of layout properties, that could be simply ignored), and which focus only on implementing the Template Layout Module.

To summarise and clarify the paragraph above, the main purpose of this alternative would be to *create a layout engine as small as possible, with a minimal CSS 2.1 support, but that implements CSS3 Template Layout Module as better as possible*.

Reusing a Layout Engine?

Another intermediate option between this alternative (implementing a completely new layout engine) and the previous one

(modifying the source code of a real browser) that was also carefully studied would be using any library that provides a certain support for HTML and CSS, or reusing the code of some small browsers, instead of starting from the scratch.

In this sense, the following "browsers" and layout engines were also studied in our review:

- Flying Saucer (2009)
- ViewML (Century Software, 2009)
- XSmiles (2008)

In my opinion, although some of these experimental browsers have a code that is relatively small and understandable, their benefits are less than its drawbacks when compared with the industrial browsers reviewed in the last section. With the sole exception of Flying Saucer, they can not be considered fully CSS 2.1 compliant, and neither their code nor their documentation is so better as to compensate for choosing one of them instead of the much more capable Firefox, WebKit, or any other of the open source industrial browsers, which also count with a much greater community of developers. Therefore, this intermediate solution was discarded, although a deeper review of their source code may still be recommendable, if only to acquire knowledge about the design of a layout engine. But, in summary, *if the alternative were to implement our own layout engine for the Template Layout Module, it should be actually from the scratch*, and not reusing code of any other CSS renderer.

Another Prototype of This Thesis

Although this was not the alternative finally chosen for the development of the prototype in our research project, it turned out to be eventually implemented out of such project (but still within the research conducted for this thesis), by a student who asked me to be the supervisor of his undergraduate thesis. By that time, the development of ALMcss, the JavaScript prototype presented in this chapter, which was the final chosen option, had already been commenced, but the idea of developing a small layout engine from the scratch still was (and *is*) very appealing to me. So I entrusted him with that task. Some months later, the result was not one, but two implementations of the Template Layout Module (then named *Ad-*

vanced Layout Module): a little Java desktop browser and a C# version of the same browser for mobile devices, developed in .NET Compact Framework (Cabal, 2006).

Both browsers not only implement the first public working draft of the Advanced Layout Module (), but they also support the following HTML elements and CSS properties.

The supported HTML elements are:

- Headings: `h1`, `h2`, `h3`, `h4`, `h5`, and `h6`
- Paragraphs: `p`
- Line breaks: `br`
- Images: `img`
- Links: `a`
- Emphasis: `em`
- Strong: `strong`
- `div`
- `span`

As for CSS, the following features are supported (in addition to the CSS3 Advanced Layout Module):

- Linking a external style sheets through the `link` element
- Class and type selectors
- Fonts: `font-family`, `font-size`, and `font-weight`
- Colors: `color`, and `background-color`
- Margins: `margin`, `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`
- Paddings: `padding`, `padding-top`, `padding-right`, `padding-bottom`, and `padding-left`
- `width` and `height`
- `display`: `inline` | `block`
- The following length units are allowed: `px`, `em`
- Percentages are also supported

The ALM mobile renderer can be seen working in figure 3, which shows two grids, as they are seen in a old PDA with Microsoft Pocket PC 2003.

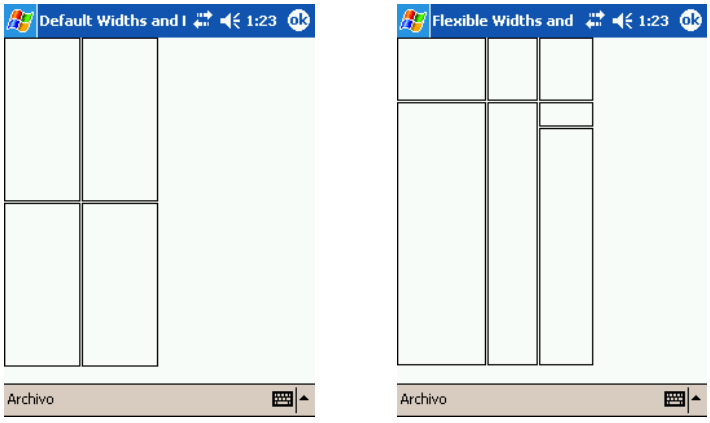


Figure 3. Two templates, as they are rendered by the prototype that was developed by Cabal (2006), under my supervision, as an alternative to ALMcss, the JavaScript prototype that will be later described in this chapter. Cabal's prototype (a small HTML/CSS/ALM renderer) is written in C# for the Microsoft .NET Compact Framework. The examples in this figure are screen captures made on an old HP iPAQ 4150 PDA running Microsoft Pocket PC 2003.

The figure on the left corresponds to the following simple grid, when it is applied to an empty document (no elements in the `body`). The template defines a simple 2 × 2 template that uses the default values for column widths and row heights ('*' in both cases) defined by the existing version of the Advanced Layout Module (Bos, 2005) when this prototype was developed:

```
body {
  display-model: "ab (*),
                  cd (*)"
                  (* *);
}
```

As for the right figure, it is the result of displaying the following template, which mix fixed and flexible heights:

```
body {
  display-model: "abc (50px),
                  def (20px),
                  ghi (*)"
                  (70px 40px);
}
```

The Java Desktop Version

In addition to the mobile renderer mentioned above, Cabal, for the same undergraduate thesis (2006), also ported it to J2SE, therefore

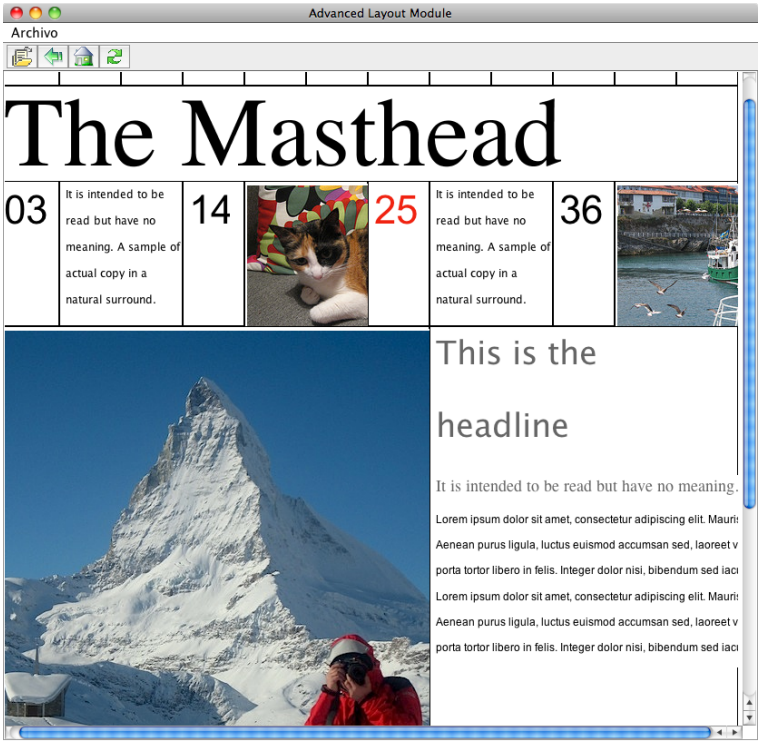


Figure 4. The desktop Java version of the Advanced Layout Module renderer.

creating a desktop version with the same capabilities (that is, limited support of HTML and CSS, plus nearly full implementation of the first Advanced Layout Module Working Draft). Figure 4 shows a screen capture of it displaying a page written in our subset of HTML with some CSS rules for fonts and colours, and template based positioning. It is a grid of twelve equal columns, with no gutters between them. A portion of the style sheet is shown in figure 5 (note that, due to parser deficiencies of the prototype and a notably absence of most CSS selectors, many classes and duplicated styles have to be used):

Conclusions

The example in figure 4 shows how a relatively complex layout can be implemented in our prototype even in the absence of floats and absolute positioning features, just using the limited template features defined in the first public draft of the solution proposed by this thesis: the Template Layout Module. And, if just the features of

```
body {
  display-model:
    "............ (50px),
    aaaaaaaaaaa (intrinsic),
    bccdeefgghii (160px),
    jjjjjjjkkkkk (intrinsic)"
    (*);
}

h1 {
  position: a;
  font-size: 3em;
  font-family: Times;
}

.one {
        position: b;
        font-size: 44px;
}

.one-text {
        position: c;
        font-family: "Lucida Sans";
}

...

.four-text    { position: i; }
.main-picture { position: j; }

h2 {
        position: k;
        font-family: "Lucida Sans";
        font-size: 36px;
        color: #666666;
}

h3 {
        position: k;
        font-family: Times;
        font-size: 18px;
        color: #666666;
}

.text { position: k; }
```

Figure 5. Part of the style sheet for the layout of figure 4. It uses a template (*grid*) that consists on twelve equal-width columns with no gap (*gutters*) between them, within several *modules* (*slots*) have been defined, each spanning a certain number of rows and columns. Each slot acts as a container where the different pieces of content of the page can be placed. This template is right rendered in our ALM desktop renderer.

grouping selectors, multiple declarations (Bos, Çelik, Hickson & Lie, 2009, §5.2.1), descendant selectors (Bos et al., 2009, §5.5) and *cascading*, were added to the renderer, such design could be created with less HTML and CSS code than it is required in any real, full featured CSS 2.1 browser. I think that this well serves as a demonstration of the goodnesses of the proposed solution.

Even with no floats nor absolute positioning features, a little renderer developed completely from the scratch is able to create complex layouts easier than in a full featured CSS 2.1 browser, and with less HTML and CSS code. This proofs both the advantages of the Template Layout Module and how relatively simple to implement it is.

In addition, as it has been shown in this section, creating from the scratch a little layout engine that implements the solution proposed in this thesis was not a so herculean effort as it could be thought at first, and, although this was not the alternative chosen for the ALMCSS research project, it was later developed in Cabal's undergraduate thesis (2006), for which both a Java desktop and a C# mobile versions of the same Advanced Layout Module (Bos, 2005) renderer were implemented.

However, this option also suffers from the lack of public visibility that has already been mentioned for the case of modifying an open CSS layout engine. Even though it would have perfectly serve as a proof of concept of this thesis, I had imposed another requirement on myself: as a member of the W3C and coauthor of the Template Layout Module, I really wanted that this thesis also served as a way of promoting the module among web designer community and convincing them of its advantages. To do so, it is needed a solution that works on the browser itself and that understands every current CSS property, and not just a subset of them. Therefore, this solution was discarded for our research project.

Implementing an Extension of an Existing Browser

Giving our needs of a relatively easy of implementation solution while in turn were as visually appealing and easy to use as possible for end users, this seemed to be one of the most attractive alternatives for developing our prototype. Since the title of this subsection may be some ambiguous, I must first clarify what I understand by a browser extension, in the context of this dissertation, specially when compared with browser plugins.

Browser plugins are binary programs that are installed by the user in the browser itself, extending its functionality (for instance, the Flash Player almost ubiquitously present in any browser if the user wants to access to YouTube videos and other similar content). Plugins generally access to internal browser structure through some API provided by each specific browser. They are also responsible for most browser crashes and security vulnerabilities (2006).

Conversely, browser extensions (also known as "add-ons"), are primarily JavaScript applications that frequently use other web technologies, like CSS and DOM. In the case of Firefox, if the extension requires user interface, this is written in XUL, the user interface language that was reviewed in a previous chapter. Firebox extensions also rely on XPCOM for accessing to some of the core components of its layout engine, Gecko. XPCOM is a cross platform component model, conceptually similar to Microsoft COM, that provides a set of core components and classes for memory management, threads, or basic data structures (2009b). By specifying the *interface* of a component using the XPIDL interface description language (IDL), XPCOM components can be written in any of the languages for which it has *bindings* (currently, JavaScript, Java, Python, Perl, and Ruby, in addition to C and C++, the implementation languages of XPCOM). Thus, a developer can access from the code of its extension to other XPCOM components as well as to expose modules of his extension to any other XPCOM component.

Two would be the main benefits of developing the Template Layout Module prototype as a browser extension (specifically, as a Firefox extension):

- The XPCOM library provides access to some of the internals of the browser, with more control than simply reading and manipulating the DOM from JavaScript, but without the complexity of modifying the source code of the layout engine.
- Extensions are very easy to install by end users. This, combined with the widely use of Firefox, would give our prototype the possibility of being used by a great number of web designers and developers to test the possibilities of the Template Layout Module.

It also has the drawback, though, that if we opted for this solution, we would be constricting ourselves to an specific browser. However, this is not too problematic, as long as a widely used browser, such as Firefox (which seems to be the most appropriate for this approach and which counts with the best documentation about the process of developing extensions) were chosen. In conclusion, this solution would be a balance between modifying an open source exist-

ing browser and developing a completely new prototype from the scratch.

Developing a JavaScript Plugin

Although the latest option of building a browser extension seemed adequate for the purposes of our research thesis and, by extension, of this thesis, it was not our final decision. While studying that alternative, we wondered whether, if the core of Firefox extensions are just standard web technologies (JavaScript, DOM, and CSS), it would be possible to implement the Template Layout Module prototype using just those technologies, as a general JavaScript plugin not tight to any particular browser.

I have to admit that I was somehow skeptical about the relatively complex layout processing that had to be done by the prototype could actually be implemented using just plain JavaScript and standard DOM, without relying on the access provided by XPCOM to the layout internals of Gecko.

However, that was until we reviewed the Edwards' IE7 project (2010). It must not be confused with the Microsoft browser of the same name (which did not exist when Edwards wrote IE7). IE7 is a JavaScript library that, in its origins, was aimed to make Microsoft Internet Explorer 6 (IE6) behaved more like a CSS compliant browser. Specifically, by linking to an HTML document the existing version of this JavaScript library when our review was conducted (Edwards, 2005), which was compound of fifteen JavaScript modules, IE6 was able to understand, among others, the following CSS properties and features:

- Advanced CSS selectors
- `min-height` and `max-height`
- PNG transparency

In addition, some well-known IE6 bugs were solved. Edwards was pioneer in using JavaScript for solving deficiencies in the CSS support of web browsers. For us, the excellent Edward's work constituted an evidence that JavaScript could be actually used to carry out

non trivial additions to the layout behaviour of web browsers, and it was the first source of inspiration for ALMcss.

While we were reviewing the alternatives for the implementation of the prototype, Savarese (2005) wrote an introductory article about other CSS3 Working Draft: the Multi-column Module. Savarese's article was accompanied by a prototype, a cross-browser JavaScript implementation of the multi-column module. Both author's works definitively showed us that a JavaScript implementation of the template layout module had to be possible.

JavaScript allows to write cross-browser extensions that add new CSS features not yet supported by browsers, and that can be used transparently to the user, just by linking the author the JavaScript code in the source HTML document.

On the other hand, writing our prototype using just standard JavaScript and DOM had a clear advantage over all other alternatives in terms of the *visibility* of the project: in theory, it should work on any browser, current or past, that supports JavaScript. Although this is not so easy in practice, due to differences in JavaScript and, above all, DOM browser implementations, it can be solved writing carefully our JavaScript code to deal with those differences.

Even better, so developed prototypes can be currently used, even in real web sites[1], without any user intervention (and even without he actually *notices* it), simply by linking the JavaScript code from the HTML document.

The major inconvenience, as it has already been stated, is the fact of having to program it in JavaScript. Not by the language itself, which, as Crockford has repeatedly said, "is the most misunderstood programming language" (Crockford, 2001), and counts with many good features, but due to some related problems that did not exist with the other alternatives, to wit:

- *Almost everything must be programmed from the scratch.* While by implementing a browser extension we could rely on the facilities provided by the underlying API, now there is a considerable amount of job to be done before implementing the functionality of the template layout itself: all style rules applied to the document must be retrieved and manually parsed; as a consequence, CSS selectors must be com-

1 This is not recommendable, though, if the JavaScript code performs some *critical* action over the final appearance or behaviour of the web page, because JavaScript support might have been deactivated by the user, even though he has a JavaScript capable browser.

pletely implemented in JavaScript to be able to retrieve the actual elements (the DOM *nodes*) for which a template is defined or that are positioned into some slot; only the computed styles of an element can be accessed from the DOM; etcetera. In summary, many low-level functionality must be implemented before we are able to concentrate on the new layout features.

- *DOM scripting is one of the more intricate parts of JavaScript programming.* It is not only that DOM itself requires verbose programming to do common tasks, but the fact that, in Resig's words (Resig, 2009), "if there is a DOM method, there is probably a problem with it somewhere, in some capacity."
- *Lack of JavaScript programming tools.* Although the situation is rapidly changing, when the development of ALMcss started, there were barely good IDEs for editing and debugging JavaScript code. It also lacked from automated testing frameworks of logging facilities of the same quality than the existing ones for mainstream programming languages like Java, C++, or C#. Even today, due to its dynamic features, it is difficult to find something as simple as a reliable documentation generator that works for every possible language construction.
- *Having to learn another programming language.* We, the research team, were much more used to program in strongly typed, static, class based object oriented programming languages like the ones mentioned in the last point. Implementing the prototype in JavaScript would force us to learn not only another syntax but, to some extent, a new programming paradigm, to get used to JavaScript advanced features like the differences between pseudoclassical, prototypical, and functional inheritance (Crockford, 2008), lexically scope closures, etcetera.

However, in my opinion, the issues outlined above are just that: inconveniences, not essential insurmountable problems of this feature. They would make us less productive implementing the prototype than we would have been in other programming languages to which we were more used, but I thought that that was a small to pay when compared with the fact of being able to have an earlier cross-browser implementation of the Template Layout Module that

could work even in the modern mobile browsers that counts with a more than decent JavaScript support. My decision, based on the review that was made for our research project with CTIC Foundation, was made: the implementation of the Template Layout Module would be developed as a cross-browser JavaScript prototype.

Once the main conclusions of our review have been discussed, the rest of the chapter is devoted to describe the architecture and design of the prototype, which was named ALMcss (*Advanced Layout Module for CSS*).

Design of the Prototype

ALMcss, the prototype that implements Template Layout Module in current browsers, is a relatively complex piece of JavaScript, with near two thousand lines of code. Since it is not possible to directly manipulate the internal layout engine of the browser (once that option has been discarded by the reasons exposed above), all the work must be done through the Document Object Model, which is the most intricacy part of JavaScript programming. In addition, there are many tedious labour to do before to actually perform the layout algorithm itself. Thus, a great portion of the code is devoted to retrieve and parse the styles applied to the document, something that is usually performed by the browser but that in this case was not possible, because the new values defined by the Template Layout Module are not yet supported by current browsers. Finally, although the situation have improved a lot since the first version of the prototype was developed, JavaScript still suffers from a lack of development environments, debugging tools, automated testing frameworks, or logging libraries of the same quality than that of mainstream programming languages like Java, C++, or C#.

Although it would be very extent to describe here all the details of its development, as it was done in the final report of our research project (Acebal, Rodríguez, García, Cueva & Labra, 2006), the main design decisions will be outlined. Other sources of information about the prototype are the undergraduate theses of the stu-

dents later involved in its refactoring (Rodríguez, 2007; Cabal, 2009).

Architecture of ALMcss

The rendering process in ALMcss is divided into four sequential processing steps, thus following a *Pipes and Filters* architectural pattern[1] (Buschmann, Meunier, Rohnert, Sommerlad & Stal, 1996, pp. 53–70). The four phases of the rendering process are enumerated and briefly described below:

The Rendering Process

Parsing the style sheet

First, all the style rules that are applied to the document are parsed, to obtain those rules that contain a template definition, a slot position, or a vertical alignment property. Those are the rules (and the elements retrieved by their selectors) that should be processed by the prototype, since they are not yet understood by browsers.

Decorating the DOM

Browsers does not store information about the CSS declarations that use any of the values defined in the Template Layout Module for `display` or `position` properties, since they are not yet officially part of CSS and therefore make the declaration *illegal* and must be ignored, according to CSS 2.1 specification. The approach followed in ALMcss to retain them until they be processed in subsequent steps is to store them in the Document Objet Model itself. It is what we have called *decorating* the DOM.

Parsing the templates and creating the object structure

The previously annotated template properties are retrieved from the DOM and parsed, creating an object structure that represents the templates defined in the style sheet: how many rows and columns

1 Actually, it uses a simplified variant of that pattern, since the different phases of the rendering process (the *filters*) are combined into a single JavaScript program, passing data each other directly, instead of being communicated by *pipes*. This is a common technique in compiler construction.

they have, what slots they contain, the position of each slot and how many rows and columns they span, etcetera.

Computing the dimensions of templates and slots

During this phase, for each template, the dimensions of their slots and the template itself are computed, according to the rules defined by the Template Layout Module that were described in *a previous chapter*.

Positioning the elements into slots

The final stage consists on actually moving each template positioned element into the appropriate slot. Templates and slots are absolute positioned according to the dimensions calculated in the previous step and they are rendered by the browser.

The four phases are shown in figure 6, which depicts the overall architecture of the prototype. Following sections describe the design of each phase in more detail.

Parsing the Style Sheet

In this first stage, all the style rules applied to the document must be parsed. This is needed to retrieve the style declarations that use the properties and values defined by the Template Layout Module. Although it reuses the existing properties `display` and `position`, they are augmented with new values specific of the Template Layout Module, and for this reason they can not be accessed through the DOM, because these new values make such declarations *illegal*[1]. Therefore, all the styles applied to the document must be *manually* fetched and then parsed to retrieve only those declarations in which we are interested: template definitions, elements positioned into a

1 The exception to the stated above is Microsoft Internet Explorer 6, which do return the value of any property for which it is asked, even if it does not know such property or its associated value. In my opinion, it would be desirable to standardise this behaviour, because it simplifies enormously the task of using JavaScript to add experimental support to new CSS properties, something that, if I am not wrong in my prediction, will become more and more common.

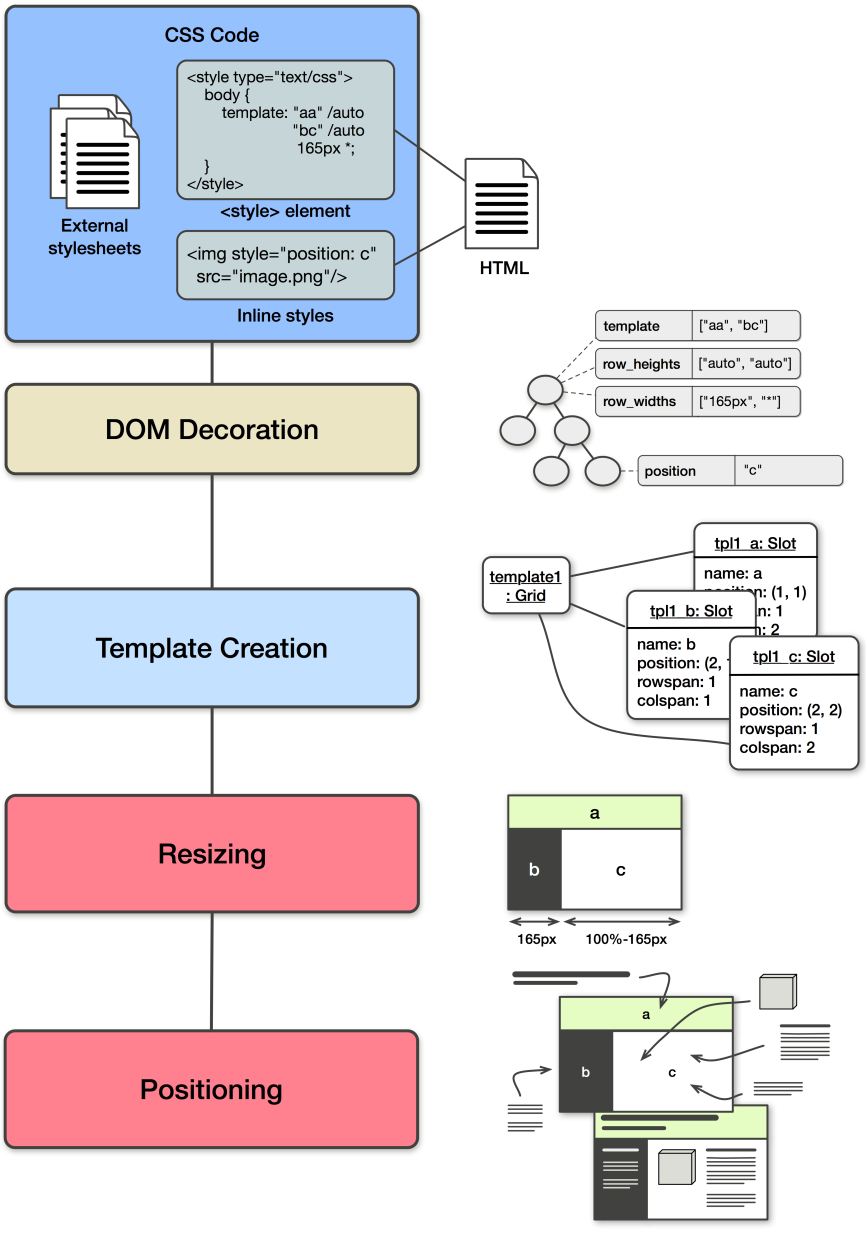


Figure 6. ALMCSS architecture consists on four sequential phases that implement the rendering process.

slot of some template, and, finally, those `vertical-align` properties that affect to some slot. The process is outlined below:

1 First, style sheets are traversed. Note that this require to fetch the CSS code of:
   - Inline styles
   - Styles embedded in the `style` element
   - External style sheets
2 Once all the CSS rules have been compiled, the resultant code is parsed, looking for the following declarations:
   - a `display` property that has a template definition as a value
   - a `position` property with a *letter*, `@`, or `same` value
   - a `vertical-align` property

Obtaining the Style Sheets

Obtaining the CSS code stored in external files have not been an easy task and it required a considerable amount of time until we were able to reach a solution. It consists on using Ajax techniques and request such external style sheets using the `XMLHttpRequest` and `ActiveXObject` objects, depending on the browser. These objects have also been required to obtain the CSS code embedded in the `style` element in the case of Opera browser, because for security reasons it can not be accessed via the DOM `innerHTML` property as with the rest browsers.

Parsing the CSS Rules

Once all the style rules that are applied to the document have been retrieved and compiled in a single string of text, they have to be parsed. Note that most rules will be simply ignored, since the prototype only cares for those declarations that make use of template positioning, namely:

- template definitions
- slot positioned elements
- vertical alignment
- `::slot(x)` pseudo-elements

But there is another issue that must be solved: since the Template Layout Module reuse existing `CSS` properties (`display` and `position`), fetching the appropriate declarations is not so simple as to just looking for such properties: if, for instance, a `display: block`

```
function createXHR() {
  var XHR = false;
  try {
    XHR= new ActiveXObject('Msxml2.XMLHTTP');
  } catch(e1) {
    try {
      XHR=new ActiveXObject('Microsoft.XMLHTTP');
    } catch (e2) {
      XHR=false;
    }
  }
  if (!XHR && (typeof XMLHttpRequest !='undefined' ||
                       window.XMLHttpRequest)) {
    XHR=new XMLHttpRequest(); return XHR;
  }
}

/* Reads an extern CSS file.
 * @param filepath the path to the file to be read
 * @return a string with contents of the file read
 */
function readCss(filepath) {
  var output;
  var request = createXHR();
  request.open("get", filepath, false);
  request.send(""); output = request.responseText;
```

Figure 7. Code for obtaining the CSS code stored in a external style sheet.

or a `position: absolute` declarations are found, they should not be processed by the prototype, since they are normal CSS 2.1 properties that must be left to the browser.

It is needed, therefore, to parse the value applied to each of these properties to determine if they belong to CSS 2.1 or are those

defined in the CSS3 Template Module. This is done in the prototype using *regular expressions*, a common technique in JavaScript.

This process is done in the methods `getPseudoCssRules` and `getPseudoElement`. The former obtains the rules that use some of the new properties of the module and the latter gets the `::slot(x)` pseudo-elements.

Decorating the DOM

Once all the style rules have been obtained and parsed, they must be stored in some place, so that they can be accessed by other modules of the prototype. Making an analogy, we could say that we need something like the *symbol table* of any compiler or interpreter (Aho, Sethi &Ullman, 1990, pp. 443–454). Given the high number of searches that are going to be done in subsequent phases of the rendering process, some associative structure, like a *hash map* would be desirable. Although JavaScript does not count with such type of structure *as is*, its arrays can perfectly emulate it, since they allow to use strings for the index of the array. Thus, it is possible to do things like:

```
var array = new Array();
array['one'] = 'first';
array['two'] = 'second';
array['three'] = 'third';
var element = array['three']; // element === 'third'
```

In our case, the access key will be the *selector* of the style rule, for which an object with information about the template properties of such rule will be stored in the corresponding position of the array.

Creation of the Structure

The next phase consists on processing the CSS rules that are using some of the template positioning properties —which have been previously stored in the DOM in the previous phase— to create the object structure that represents the templates and their slots. Note that this phase does not make any actual layout process: it simply

recreates in memory the same structure of templates that have been defined by the user in the style sheet.

This phase is divided into two steps:

1. First, the previously decorated DOM is parsed to identify all the templates defined in the style sheet, and a structure of JavaScript and DOM objects are created for representing such templates.
2. A second step actually moves the elements of the original HTML document into their corresponding slot (one of the DOM objects created in the previous step).

To understand how this process works, let us consider the following template definition in CSS:

```
display: ". a b" /165px
         "d e b" /auto
         "d c c" /220px
         (180px * 12em);
```

A template can be represented with a two-dimensional array. Once the raw data of the template definition (the value of the `display` property) have been dumped into the array, the next step is to identify the slots that compound the template. First, a `Grid` object is created and associated to the actual HTML element where the template is defined (a DOM `HTMLElement` object). Then, the array that contains the template definition is traversed, according to the following algorithm:

1. The template is traversed, starting in the position (1, 1) (first *row*, first *column*), and following from left to right and from top to bottom ((1, 2), (1, 3), (2, 1) ...).
2. A `Slot` object is created and assigned to the template (the `Grid` object) with the current position.
3. While the slot identifier matches that of the last position, the `colspan` or `rowspan` properties of the current slot are incremented accordingly.
4. When a different slot identifier is found, a new `Slot` object is created.

As for the creation of the templates themselves, it deserves more explanation, because each `Grid` object is not isolated, but it must know its *ancestors* (in compiler construction terms, its *scope*). This is needed for the case of *nested templates*. The pseudocode is as follows:

```
if (current DOM element is a template)
  // Creates a new Grid object with no position and
  // width and height equal to 0
  var grid = new Grid()
  grid.makeGrid() // Creates the slots of the template
  if (the element does not have any ancestor)
    add it to the list of containers
  else
    add it to the ancestor
  end_if
end_if
// Parse the children of the current element passing
// this element as ancestor
foreach (child: element.getChildren())
  child.parse(element)
end_foreach
```

The whole process is shown in the UML sequence diagram of figure 8, where the interactions at execution time among the different objects are outlined, for an imaginary template definition like `"aaab"`, `"aaac"`....

Creation of HTML Elements for Slots

Another step consists on creating an HTML element for each slot in the template. This is done through the DOM. These `HTMLElement` nodes act as placeholders to contain the actual elements of the HTML document that are positioned into slots, which will be done in the next phase of the rendering algorithm. Note that this step would not be necessary if we were implementing the Template Layout Module natively in a browser. In that case, *boxes*, and not actual elements, would be created to represent the slots. But, since this is not possible from JavaScript, the internal boxes of the layout engine are emulated inserting "artificial" HTML elements into the Document Object Model.

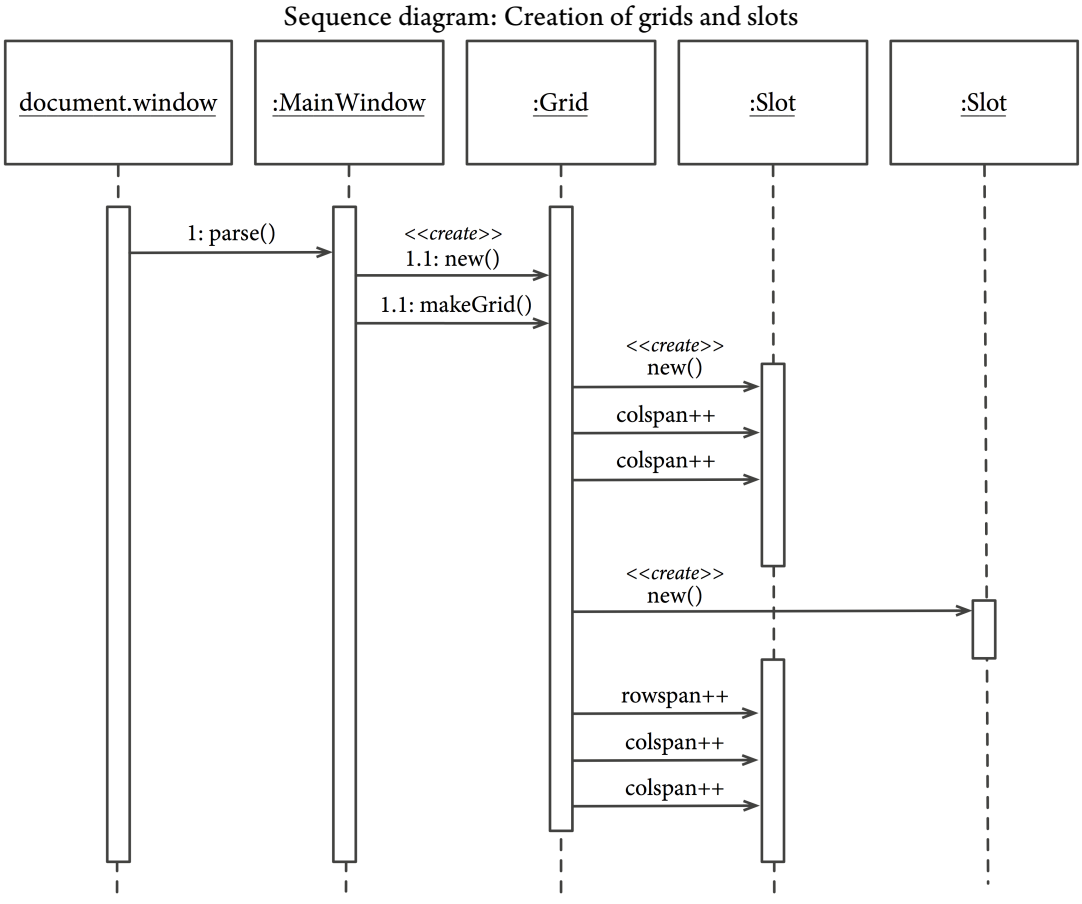


Figure 8. **UML sequence diagram showing a possible scenario of creation of a grid. First, the `a` slot is created, and since it spans three columns, its `colspan` property is incremented. Then, in position (1,4) a new slot identifier is found: `b`, which does not span any columns. After the first row is processed, comes the second one. Again, the `a` letter in the position (2,1) of the template, so the `rowspan` property of the previously created slot must be incremented. The process continues until the last position of the template definition has been reached.**

Since this is the basis for what will come later, and a fundamental piece in the design of ALMcss, it will be explained with a "real" example. Let it be the HTML and CSS code of figure 9, where two nested templates are created. Then, the result of this step for that example is the creation of the structure of templates (*grids*) and slots shown in figure 10, where each grid has a reference to the HTML

Original HTML code

```
<div id="header">...</div>
<div id="content">
   <div id="nav">...</div>
   <div id="mainContent">...</div>
</div>
<div id="footer">...</div>
```

CSS code

```
body {
  display-model: "a"
                 "b"
                 "c";
}
#header { position: a; }
#content {
   position: b;
   display: "de";
}
#footer { position: c; }
```

Figure 9. The code on the left shows a portion of the original HTML document. On the right, the figure shows how those HTML elements are positioned into two nested templates. Note that although the example, for clarity, uses different letters for the nested template, this is not required, and implementations must be aware of the context of the element (its ancestor templates, that is, the *scope* where it is defined).

element where it has been defined and each slot points to the new created HTML element. At this point, the DOM for the current example would be as follows:

```
<div id="header">...</div>
<div id="content">
  <div id="nav">...</div>
  <div id="mainContent">...</div>
  <div id="tpl2_slot_d">...</div>
  <div id="tpl2_slot_e">...</div>
</div>
<div id="footer">...</div>
<div id="tpl1_slot_a">...</div>
<div id="tpl1_slot_b">...</div>
<div id="tpl1_slot_c">...</div>
```

Moving Elements into Slots

The final step of this phase consists on actually moving each element in the original HTML document to the slot where it has been posi-

Object structure of grids, slots, and HTML elements

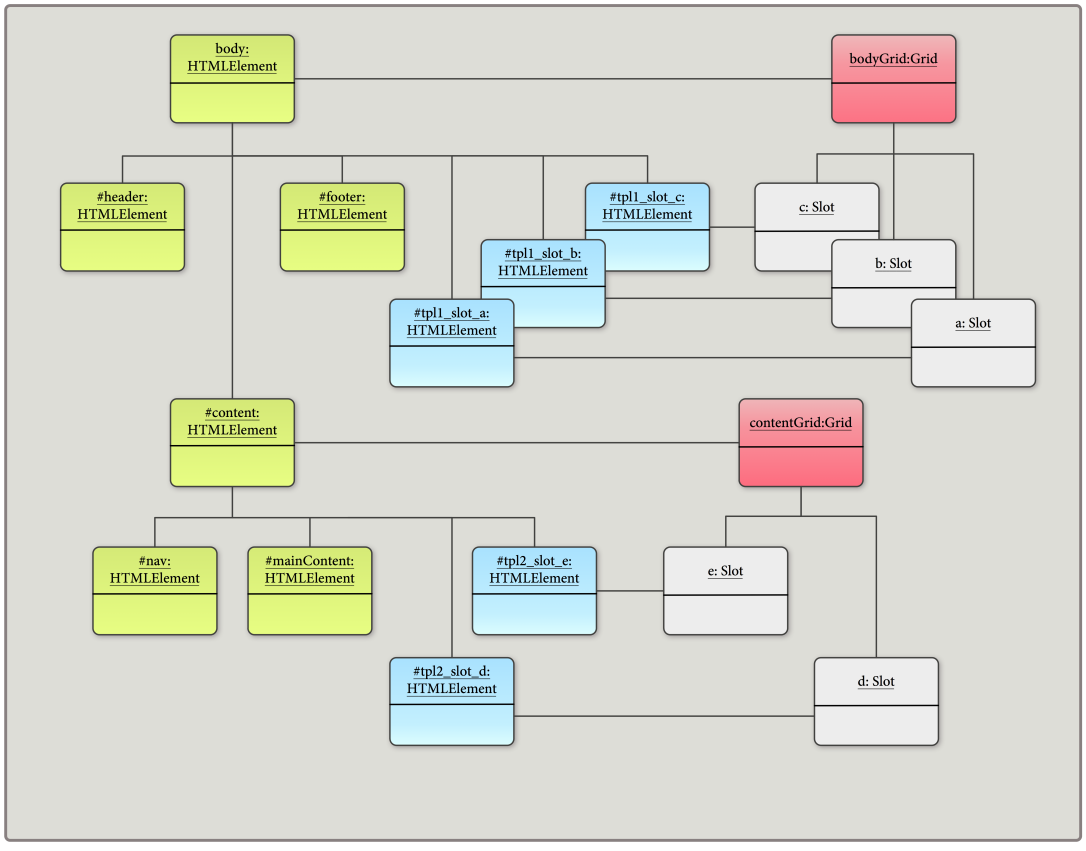


Figure 10. When the second phase of the rendering algorithm ends, an object structure of grids, slots, and HTML elements represents all the templates defined in the style sheets for a given document. The figure shows the object diagram for the HTML and CSS code of figure 9, where green boxes represent the DOM nodes (`HTMLElement` objects) of the original HTML document (they are created by the browser when the document is loaded), whereas blue ones are those `HTMLElement` that have been created and inserted in the DOM by ALMCSS to represent each slot. As it can be seen, each `Slot` object has an associated `HTMLElement`. It is in these artificially created elements where the DOM nodes that represent actual content of the document will be moved in the last phase of the rendering process. As it can be seen, for grids no elements are created, but they are associated instead with the existing DOM node corresponding to the element for which they are defined.

Object structure after HTML elements have been moved into slots

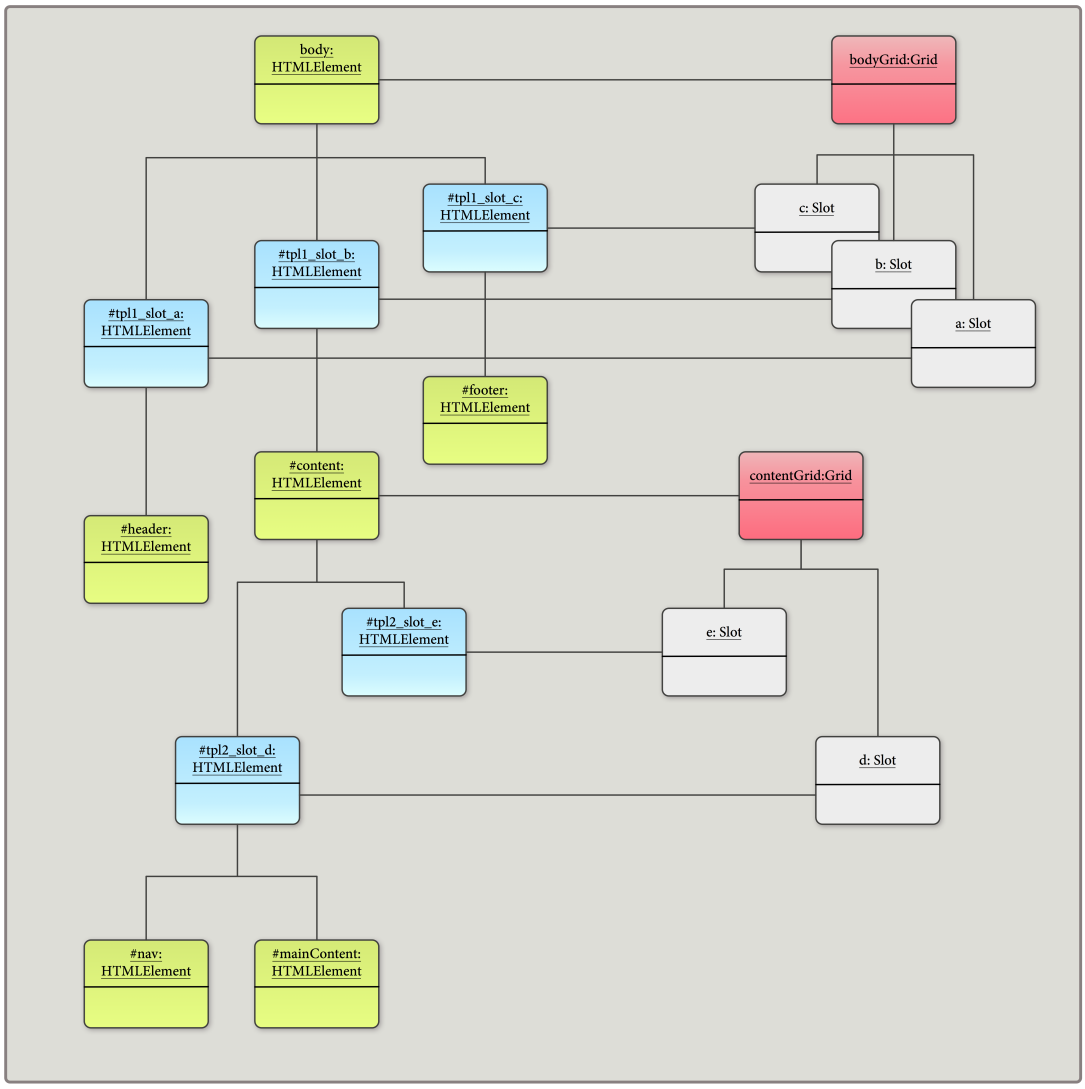


Figure 11. The final object structure of this phase, one every positioned HTML element in the original document have been moved, manipulating the DOM into their corresponding slot.

tioned (that is, to one of the `HTMLElement` nodes inserted in the DOM in the last step). The resultant structure of DOM (`HTMLElement`) and ALMcss (`Grid` and `Slot`) objects is shown in figure 11, and is represented below:

```
<div id="tpl1_slot_a">
  <div id="header">...</div>
</div>
<div id="tpl1_slot_b">
  <div id="content">
    <div id="tpl2_slot_d">
      <div id="nav">...</div>
    </div>
    <div id="tpl2_slot_e">
      <div id="mainContent">...</div>
    </div>
  </div>
</div>
<div id="tpl1_slot_c">
  <div id="footer">...</div>
</div>
```

Class Structure for Representing Templates

After finishing the creation of all the templates defined in the style sheet, a structure of `Grid` and `Slot` objects is created in memory for each template of the document. When modelling the representation of the templates, the issue of *nested templates* arose. Basically, it means that a template can contain not only slots, but also other templates inside. One of the premises of ALMcss (and of any piece of software) is that it should have the conditional logic reduced to a minimum. In particular, we did not want to have to check, every time an element is rendered, whether it is a template (a *grid*) or a slot. Fortunately, this is a well-known software design problem, as is its solution: the *Composite* design pattern (Gamma, Helm, Johnson & Vlissides, 1995, p. 163):

> *Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.*

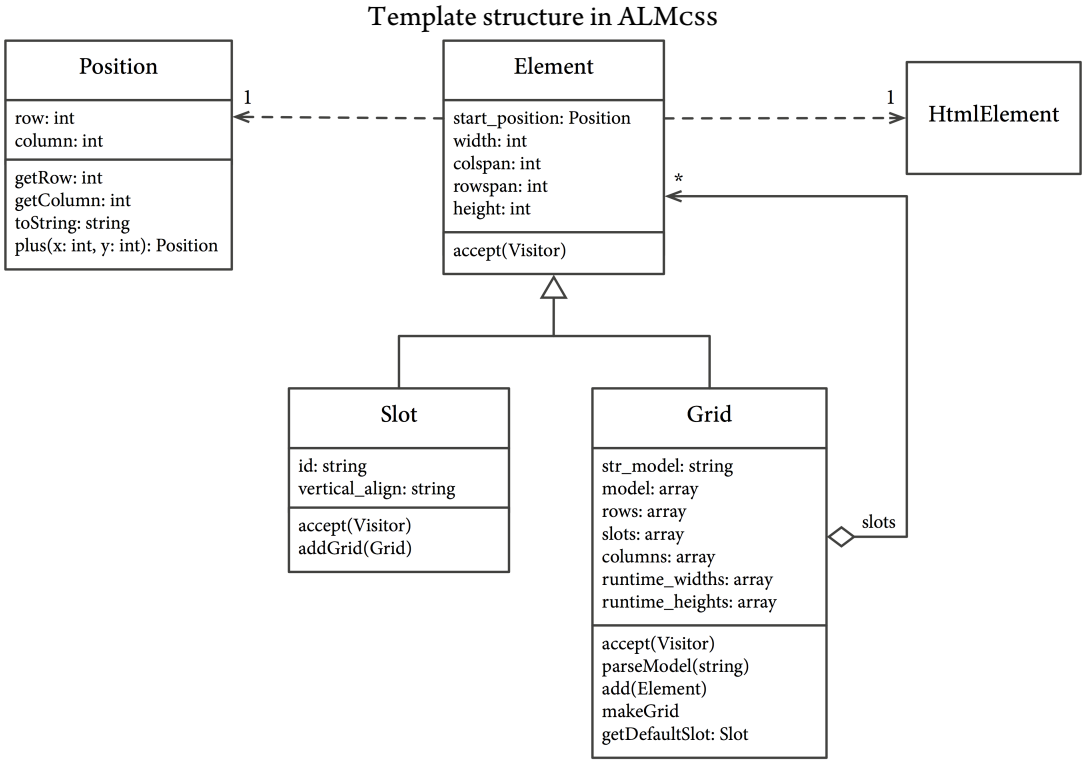


Figure 12. Class diagram showing the template structure as is represented in ALMcss.

Figure 12 shows all the classes that participate in the template representation. Specifically, the *Composite* design pattern is implemented by `Element`, `Slot`, and `Grid`, which play the roles of *Component*, *Leaf*, and *Composite* in the pattern, respectively. Note also how each element (slot or grid) created by ALMcss keeps an one-to-one relationship with the actual DOM HTML element to which it belongs.

Resizing

The third of the four phases of the rendering process is responsible for computing the dimensions of the templates and slots created in the preceding phase. This process has necessarily to be done in two steps: first, the widths are computed, and then the heights. This is so because of the *flexible* widths introduced in the Template Layout

Module with the values `intrinsic`[1] and * (*asterisk*), which were not present in the first drafts of the module (only *fixed* widths were allowed).

With these new values, the width of a slot may depend on its contents, and so do its height, if a value of * or `auto` have been specified for some row. For this reason, the dimensions of the slots (and, therefore, of the templates themselves) can not be longer computed in a single-pass algorithm, but the two passes above mentioned are needed.

For computing the size of the slots and templates, the *Visitor* design pattern (Gamma et al., pp. 331–349) has been applied. Thus, the following two visitors traverse the object structure of each template (its slots and nested templates, if any), computing their widths and heights, respectively:

- `SizingWidthVisitor`
- `SizingHeightVisitor`

Positioning

Once all the templates, slots, and their associated HTML elements, have been created, inserted in the right place in the DOM, and their dimensions have been computed, it is time to proceed to actually distribute the elements on the screen. This is what the last phase of the prototype does, the arrangement of the elements in their right position on the screen. In other words, it is in this phase where the actual *layout* of the document is carried on.

As it is natural, this phase relies on the values that have been computed in the preceding phases, specially the dimensions of the templates and slots. Therefore, this final procedure is relatively simple, since it only have to access the DOM elements that define a template and, for each, traverse the slots it contains, and read their

1 `intrinsic` has been actually removed from the current version of the specification and replaced by `fit-content`, `min-content`, `max-content`, and `minmax(p,q)`, but they are not yet supported by ALMCSS. Therefore, the explanation for the process of computing the widths will refer to the old value of `intrinsic`. Anyway, it has the same meaning than the current `fit-content` value, which has already been explained in the chapter about the Template Layout Module).

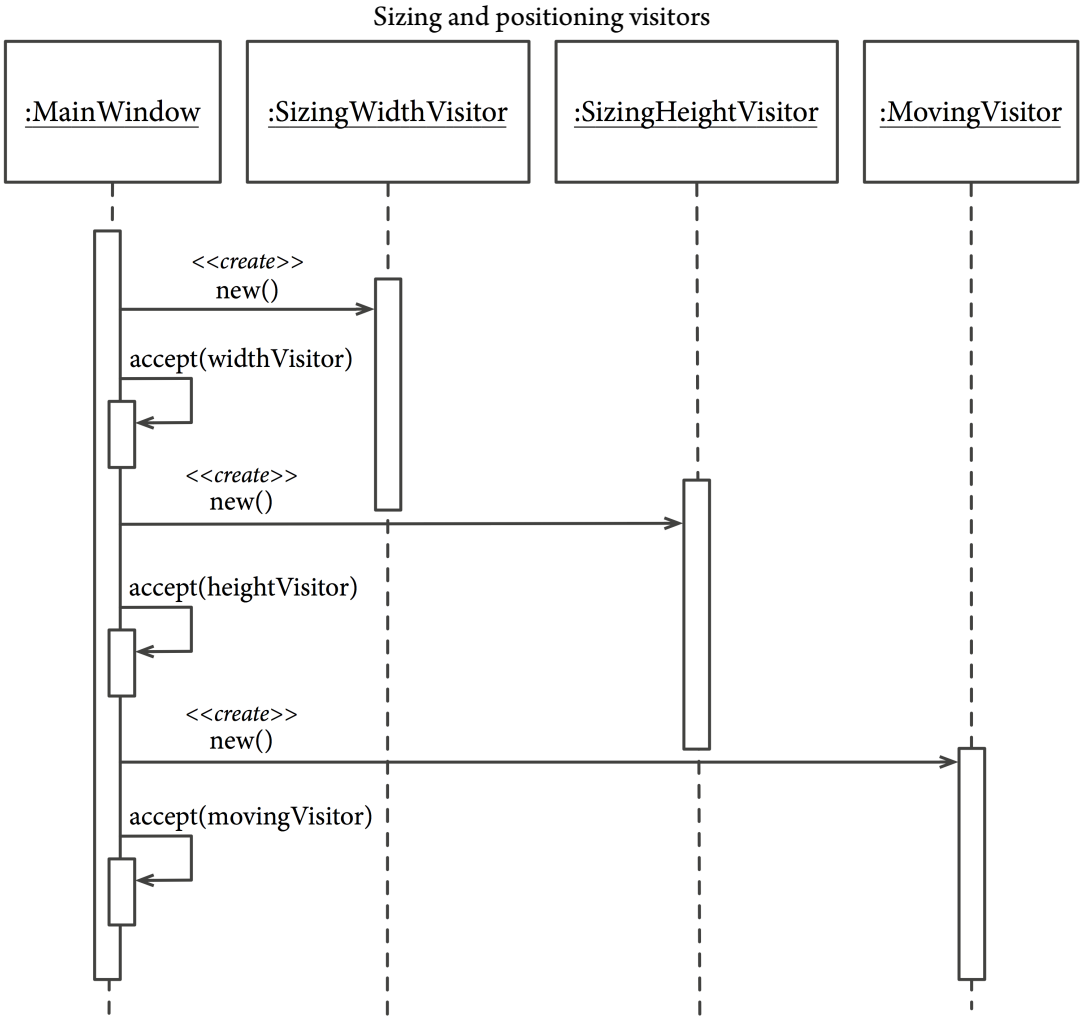


Figure 13. A simplified UML sequence diagram which depicts the sizing and positioning phases of the rendering process done by ALMCSS. It is simplified because it does not show, for clarity reasons, how the actual object structure of a template is traversed by visitors.

dimensions. As the slots of each template are being traversed, their horizontal and vertical offsets are computed based on the position and size of the preceding slot. Finally, slots are *absolute positioned* using the coordinates computed in this phase.

As it was done in the previous phase, positioning of the elements is implemented following the *Visitor* design pattern, by the `MovingVisitor` class, as is depicted in figure 13, which shows a very simplified sequence diagram of the last two phases of the algorithm.

Conclusions

In this chapter, the architecture and design of ALMcss has been presented. Its development started four years ago, when the CSS3 Advanced Layout Module that was later renamed as Template Layout Module was only a W3C "Members Only" Working Draft, and the first version was available a few months later. It was therefore the first implementation of the currently named Template Layout Module, and it remained as the only available implementation until the already mentioned recent work of Deveria (2009).

Although it is just a prototype, and it has several well-known bugs, it has served to the following purposes:

- Having a earlier implementation, even a not fully compliant one, has allowed to present the Template Layout Module using "real" examples and show them working in current browsers, thence helping us promoting the benefits of the proposed solution among the web designer community.
- It has showed that, if has been possible to implement the Template Layout Module in a current browser using just JavaScript and the DOM, without having access to the underlying layout engine, it should be feasible to implement natively by browser vendors.
- As a marginal achievement, it has proved that JavaScript can be used as a rapid prototyping tool for adding support for experimental CSS modules to current browsers, even years before than those features are natively implemented.

The design and coding style of our prototype, though, has errors. Those errors were present in the first beta releases and in the final version presented at the end of the research project (Acebal, Rodríguez, García, Cueva & Labra, 2006), and, although it was later refactored in the B.S. and master theses of Rodríguez (2007) and

Cabal (2009), the pitfalls of the initial design have burdened the development of the prototype since then. And its coding style still needs to be improved to look more like JavaScript code and benefit from some of its good features and expressiveness. All of that has caused that solving the errors of the layout algorithms be a more complicated task than it should be.

It must be noted, though, that when its development started, none of the good JavaScript books that have appeared since then (Flanagan, 2006; Keith, 2006 Adams et al., 2007; Yank & Adams, 2007; Keith, 2006; Crockford, 2008) existed, and the existing ones did not focus on JavaScript as a programming language nor describe in detail its prototype-based nature. At least, Zakas (2005) did a decent job explaining some of the paradigms for creating objects (pp. 90–98) and a whole chapter to inheritance in JavaScript (pp. 103–124), so we took it as a reference, as well as Crackford's essays and presentations (Crockford, n.d.). There also were the Keith's book on DOM scripting (2005), but it is oriented toward a much superficial DOM programming than what had to be done in ALMcss. Sambells and Gustafson's work (2007) is much more related with the DOM treatment that is done in ALMcss, but, again, it appeared two years after the first version of our prototype had been released.

In addition, today widespread JavaScript frameworks and toolkits like Prototype (2009), JQuery (2010), or YUI (Yahoo!, 2009) had not appeared or they were in their origins, and we only had the already mentioned Edward's library (Edwards, 2005) to be taken as a reference. Nowadays it is very common to use JavaScript for improving the CSS support of web browsers or adding experimental features, but our prototype was one of the first examples of this trend.

For all those reasons, I am redesigning and reimplementing the prototype completely from the scratch, to take advantage of the experience gained from its prior development. The objective is not only to have a prototype that is fully compliant with the Template Layout Module specification, solving its current bugs and implementing the lacking features, but to have it with a clean design that

allows me to experiment with possible new features not yet adopted by the W3C CSS Working Group, or to use it as the basis for further developments, like the possibility of dynamically change the grid from the browser itself, etcetera.

In addition to ALMcss, other two prototypes developed for this thesis have also been introduced in this chapter (Cabal, 2006). They consist on a mobile layout engine and its desktop version, developed from the scratch in Java and C#, respectively. As it has been shown in figure 5 on 269, they are also able to render relatively complex grids defined with the Template Layout Module, so this option will also be considered for further development, since the possibilities offered by having a complete layout engine are enormous for future research (for instance, enabling CSS *debugging* capabilities, allowing the designer to arrange content dynamically, adding complex layout features like non rectangular boxes, etcetera).

12

A Visual Layout Generator

There have traditionally been a lack of good visual editors for CSS, specially for creating layouts. Although sophisticated WYSIWYG tools exist, and they have been improved to be "standard compliant", the code that they generate is still much more untidy than its equivalent hand made.

Other tools, known as layout generators, are appearing, but they are less more than a set of predefined templates, customisables only to a certain extent.

This chapter presents another prototype created for this thesis which aims to prove that the proposed solution simplifies so much the layout process in CSS that layouts can even be created using a visual tool.

Introduction

Creating pure CSS layouts has been always burdened by the lack of good visual editors. Although WYSIWYG tools exist since many years ago, they have been always characterised by generating a polluted code that is difficult to maintain. This is not a fault of the existing tools, but an evidence of the complexity of current CSS layout mechanisms. It is very difficult, if not impossible, for an authoring tool to figure out how to translate the arrangement of elements made by the user on the screen to the low-level layout mechanisms of Cascading Style Sheets that have been reviewed in the first part of this dissertation.

That is the reason why these tools used to rely on HTML table-based layouts. Although the situation has substantially improved with the most recent versions of tools like Adobe Dreamweaver[1] or Microsoft Expression Web[2], and now they are able to generate pure CSS layouts, they tend to abuse of absolute positioning, as well as *classitis* and *divitis* (Zeldman & Marcotte, 2010, p. 160):

> *Even the best, most sophisticated visual web editors tend to cough up needless classes like so many cold germs —primarily because they are visual editors, not people. … Even when using as sophisticated a tool as Adobe Dreamweaver or Microsoft Expression, you'll want to edit its output to avoid classitis and divitis.*

Nevertheless, the Template Layout Module proposed in this thesis as a solution to the problem of layout on the web makes possible to create visual editors that are able to generate layouts without touching the underlying HTML document, creating a CSS code very similar to what would have been coded by hand.

This chapter introduces another prototype developed for this thesis: a visual layout generator that creates templates as defined in the CSS3 Template Layout Module (Abella, 2009).

1 http://www.adobe.com/products/dreamweaver/
2 http://www.microsoft.com/Expression/products/Web_Overview.aspx

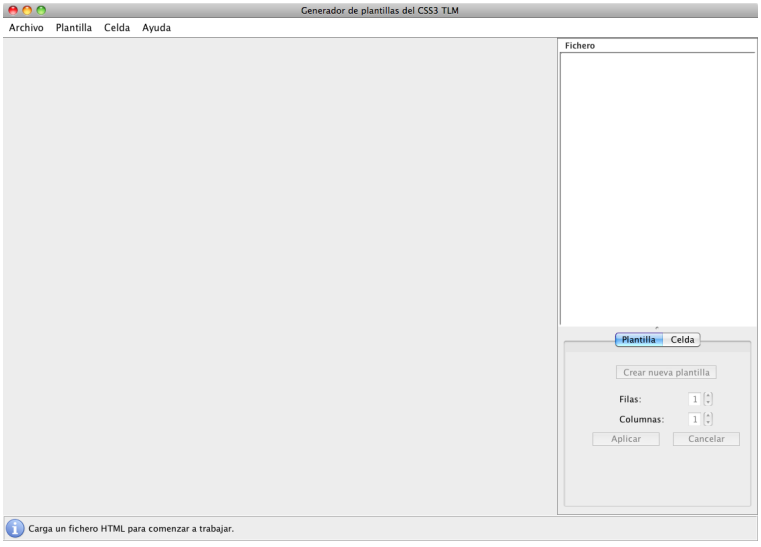


Figure 1. Main screen of the visual template generator. It is made up of three panels: the big left panel will contain a visual representation of the template that is being created; on the right top panel a tree representation of the HTML will be shown, once the document has been opened; finally, on the right bottom corner of the main screen there are the controls that allow the user to create new templates and combining and dividing the slots of the template.

User Interface

This section summarily describes the user interface of the prototype. As it is depicted in figure 1, the main screen is divided into three areas. These areas and their function are described below.

Template area

It is the big panel on the left, which will contain a visual representation of the template being generated: how many rows and columns it has, and the slots that it contains, as well as the positions occupied by each slot (that is, how many rows and columns it spans).

Document tree

On the right top area of the screen, a tree representation of the HTML document will be shown, once it has been opened in the editor.

Template and slot properties

The final panel, on the right bottom of the screen, shows the properties of the template and allows to modify them: number of rows and columns of the template, and which cells compound a slot. Cur-

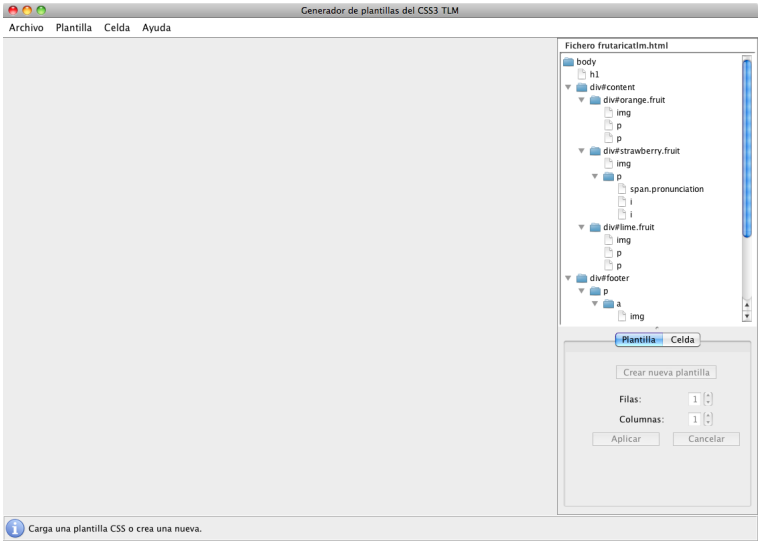


Figure 2. Once opened an HTML document, its structure is shown as a tree in the upper right panel of the main window.

rently, the prototype does not allow to modify the height and width of rows and columns (it is simply a student's undergraduate thesis developed as a proof of concept), although it would be very easy to add this feature.

Usage Example

To illustrate how it works, I am going to recreate the same example used on *Chapter 6* to explain the One True Layout technique (see p. 122), and that was later reused on *Chapter 10* (p. 246).

Opening the HTML Document

The first step is to open the HTML with which we are to work. Once loaded, its structure it is shown on the right upper panel (see figure 2).

Creating a New Template

Then, a new template must be created (figure 3). Once the number of rows and heights have been specified, the prototype sketches the

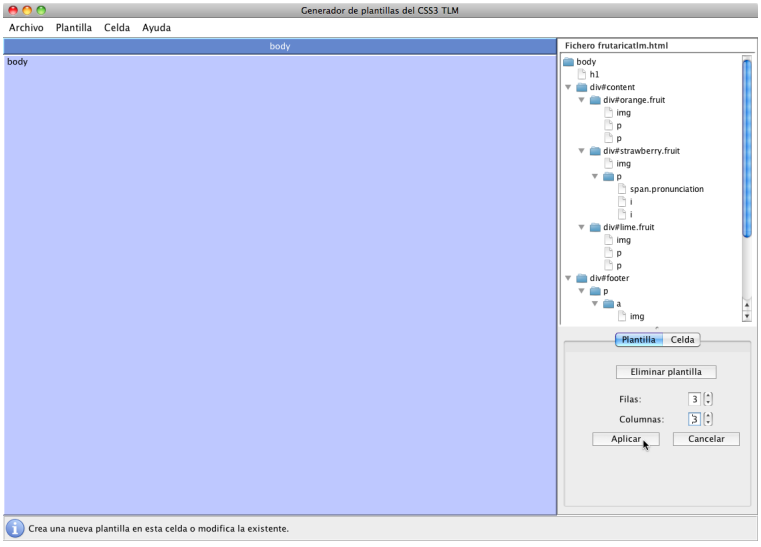


Figure 3. First, a new template is created for the `body` element.

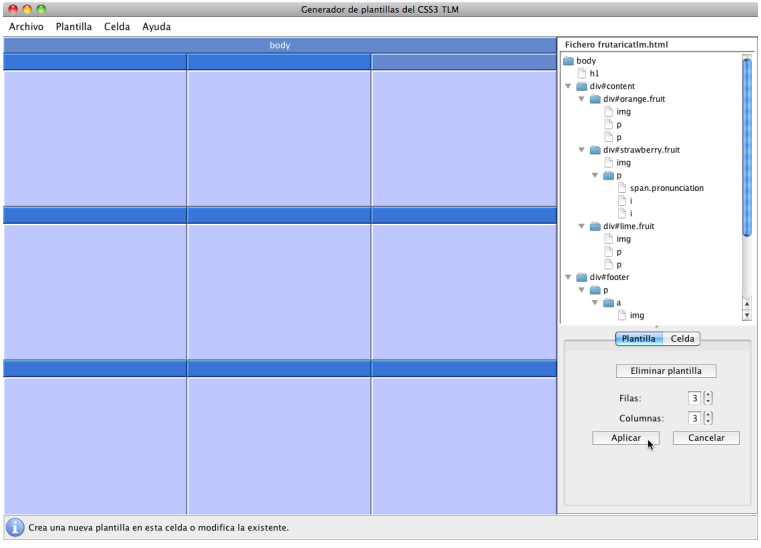


Figure 4. After specifying the number of rows and columns, the prototype shows the structure of the template on the large panel on the left.

visual structure of the template. For example, for a 3 × 3 template, the result is shown in figure 4.

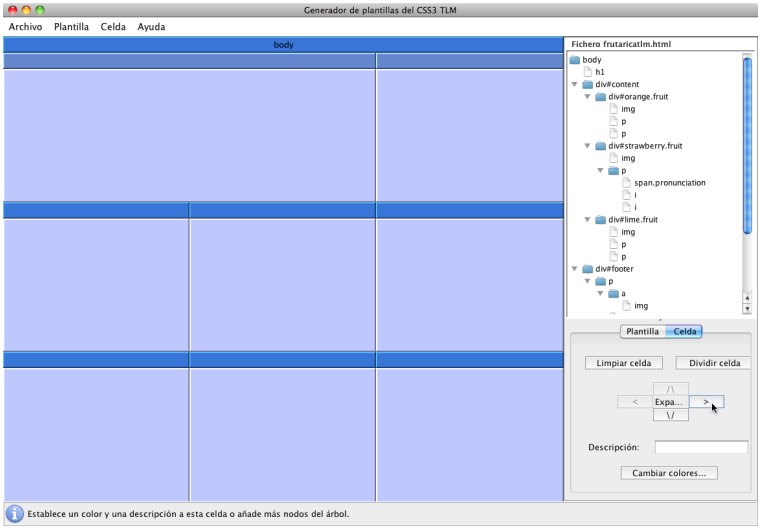


Figure 5. The user can make slots spans several rows and columns, by combining or dividing them. In this figure, the slot in the position $(1, 1)$ has been combined with that of position $(1, 2)$ to form a single slot.

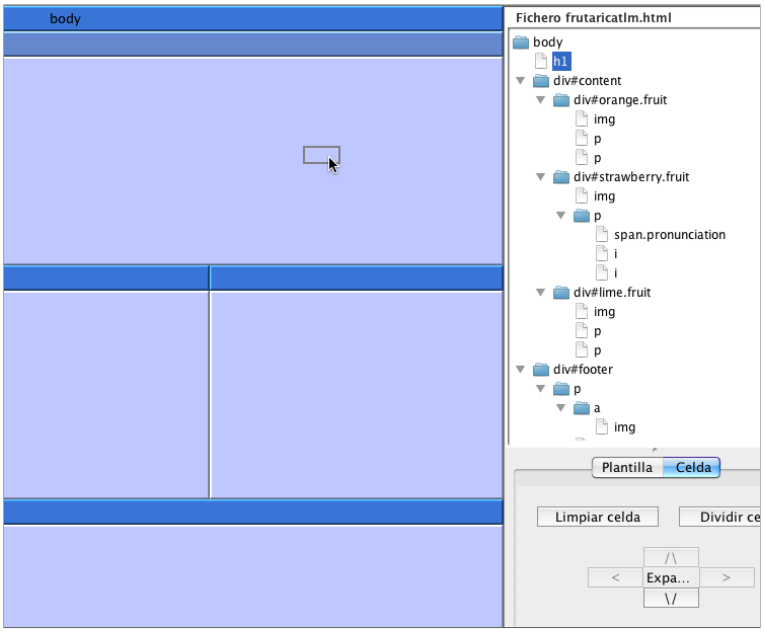


Figure 6. Template Layout Module makes possible to arrange the content in the layout simply by dragging elements from the HTML document structure and dropping them into the desired slot of the template.

Defining Slots

Although the template has already been defined, and therefore its slots created, the user can combine and divide them to change the layout (this has not necessarily to be done now: the dimensions of the slots —the number of rows and columns that they span— can be changed at any moment during the layout process, even if content had been already added to the slot).

For my example, I am going to make that first and third rows contain a single slot that spans three columns. Figure 5 shows how the process of spanning a slot is.

Arranging the Content into Slots

Now begins the interesting part. Until now it has been not more than a wizard similar to those of word processors for creating tables[1]. It is time to actually arrange the content in the layout created so far.

Thanks to the Template Layout Module, this is as simply as to drag and drop an element from the tree on the right to the slot into which it has to be positioned, as shown in figure 6. Once all the content has been laid out into the template, the final appearance of the prototype is depicted in figure 7.

Generated Template

Finally, the generated template can be saved as an external style sheet, which content is shown in the listing of figure 8.

1 Nevertheless, there is nothing similar currently to create pure CSS layouts. Naturally, as it has been mentioned in the introduction of this chapter, this is not merit of the tool —otherwise, a mere prototype very easy to implement, although meritorious for being an undergraduate thesis—, but of the Template Layout Module, which, by making the layout *explicit*, also makes it very ease to automatise. Thus, even though the prototype, in its current state, may be anything but a wizard… how would we wish to have a similar wizard today for CSS layouts! Similar commercial tools, like CSS Sculptor (WebAssist, 2009) or CSS Layout Magic (Project Seven, n.d.) are much more limited than this prototype, and they are simple sets of predefined templates of one, two, or three columns, that may been partially customised (to include or not header and footer, changing the colours, etcetera): current CSS layout capabilities do not allow more than that.

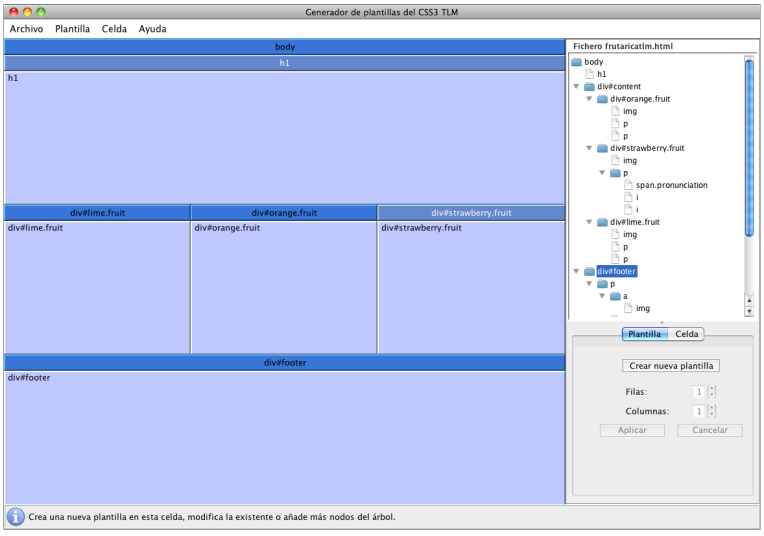


Figure 7. Once all the content has been laid out into slots, this is the appearance of the template.

CONCLUSIONS

Although the tool presented in this chapter is extremely simple in terms of functionality, I think that it serves well to the purpose of exemplifying how easy will be to automatise the process of creating layouts by visual editors and authoring tools.

```
body {
        display:
        "aaa"
        "bcd"
        "eee";
}
body h1{
        position: a;
}
div#orange {
        position: b;
}
div#strawberry {
        position: c;
}
div#lime {
        position: d;
}
div#footer {
        position: e;
}
```

Figure 8. The final template generated by the prototype.

13

Conclusions and Further Research

The last chapter of this thesis summarises, in a compressed form, which are, in my opinion, its main contributions to the issue of layout on the web. First, a very succinct review of the whole thesis is provided as an introduction to the chapter. Finally, those areas where I think that there is still much room for further research are enumerated and briefly described.

Review

This dissertation started stating the hypothesis that CSS is not a layout language. Notwithstanding the commonly accepted assumption that Cascading Style Sheets allows the separation of presentation and content, I argue that such separation is not currently possible. Neither floats nor absolute positioning nor the rest of CSS properties that deal with the box and visual formatting models can be considered true layout mechanisms. As a consequence, the layout tends to be dependent, to a greater or lesser extent, on the markup of the document, which breaks the promised separation between presentation and content.

Furthermore, creating layouts with CSS is far from being an easy task: floats, absolute and relative positioning, and margins, are low-level layout mechanisms that operate at the level of elements. Thus, the final layout is the result of the concurring interactions among all the elements of the page, and the rules that govern this interactions, as described in CSS 2.1 specification, are too complex.

This thesis has addressed the problem of layout on the web following a totally different approach to what has been done until now in CSS: instead of creating the layout on a element basis, it is now defined **explicitly**, at a high level of abstraction, using the same concepts that graphic designers use in traditional printed media: rows and columns, and the modules formed by combining several rows and columns. This means a drastic change in the way that layout is specified in Cascading Style Sheets.

The layouts thus created are not only much more easy to achieve than with the traditional CSS layout mechanisms, but they are also independent of the document structure: no matter the position that an element occupies in the document source code, it is going to be able to be arranged into any position of the layout, thus providing a sort of **content reorder mechanism**, which has proved vital for obtaining a true separation between presentation and content, or, more specifically, between the structure of the document and its visual layout.

Major Contributions

This section summarises in a compressed form which are, in my opinion, the main contributions of this thesis.

CSS Is Not a Layout Language

The first contribution is the very hypothesis of this thesis, expressed in any of its variants: "CSS is not a layout language", "separation between presentation and content is not currently possible on the web", "Cascading Style Sheets lack true layout mechanisms" "redesign is not possible with CSS"... are all of them bold affirmations, which go against established truth. Although it is true that some authors have *relieved* against this assumption, it still is a quite *politically incorrect* statement. I believe that it has been, though, sufficiently proven in this dissertation and, even if only for that —if I have been able to demonstrate that more advanced layout mechanisms are needed in CSS, be the solution proposed by this thesis or others—, I think that this thesis would have been worth it.

CSS Is a Low-Level Language

I find particularly interesting the analogy made on *Chapter 8* (190) between Cascading Style Sheets and programming languages, attending to the level of abstraction. I state that creating layouts with CSS is like pretending to develop a complex piece of software in Assembly code, and that more high-level layouts mechanisms are needed.

Implicit vs. Explicit Layout

Another original contribution of this thesis is the distinction made between implicit and explicit layout. By *implicit* layout I designate the way in which layouts are currently created in CSS, as it has been mentioned in the introductory review of this chapter. Not only is difficult to create the layout of a page with current CSS layout mechanisms: it is even more complicated to take an HTML document and its style sheet and try to figure out what layout is being applied just by reading the CSS. Conversely, I defend that a *explicit* manner of defining the layout of a document is needed, where an author can

say that the page has, for instance, three columns, a header, and a footer, and that the middle column is liquid whereas the widths of first and third columns are 13 em and 225 pixels, respectively, and that he can then arrange the elements simply by saying what elements go into which of the previously defined regions.

Cascading Style Sheets Calls for a Distinction Between Presentation and Layout

This is also an unconventional distinction made by this thesis: I have distinguished the "layout" from the rest of stylistic effects. Usually they are all of them included in the the term "presentation", but, while Cascading Style Sheets works reasonably well for most those other stylistic effects, and it has done a great job taking most of the presentational aspects out of HTML, the same is not true for layout. By distinguishing the latter as a subset of what is commonly intended by *presentation* I have made possible to focus on this specific problem, both conceptually and then proposing a solution.

Requirements of CSS Layout

The first part of the dissertation has focused on analysing why Cascading Style Sheets does not work for layout, and it has been demonstrated with a few selected case studies. The problem statement on *Chapter 8* summarises those problems and sets the two requirements that any proposal of layout addition to CSS should fulfil, to wit:

It has to provide a content reordering mechanism

That is, it must be completely independent on the order and structure of the document source code, and any element have to be able to be laid out into any position, regardless where it has been defined in the HTML. Naturally, this must be done without breaking the architectural requirements of the web, namely, the ability of the page to be rendered on any device, no matter its screen size or installed fonts.

Layout has to be explicitly defined

As it has already been explained, one of the main problems of how layout is currently done in CSS is that it is only implicitly defined by many low-level properties applied to single elements. Instead, layout should be explicit: this page will have this and that regions, with this specific dimensions —which must be able to be defined in terms of other layout elements—, and here will go these pieces of content, while those others will go into that region.

An Innovative Layout Mechanism is Proposed

This is probably the most obvious contribution made by this thesis: the Template Layout Module that has been proposed on *Chapter 9* as a solution to the problem of layout on web. However, as it already was stated in that chapter, I can not pretend to appropriate of others' work: being this a solution developed within the W3C CSS Working Group (W3C CSS-WG), it is, by definition, the result of many others' efforts, particularly Bert Bos, advisor of this thesis, who had already have been defined the basic syntax and behaviour of the formerly named Advanced Layout Module when I began this thesis and joined the W3C CSS Working Group.

One of my own contributions to the current version of CSS3 Template Layout Module is the ability to apply style to the slot themselves (the `::slot(x)` pseudo-element). Others are less visible, but still important, such as defining case studies to analyse the viability of our ideas for the module. For instance, I suggested that the default height for rows should be `auto` (formerly named `intrinsic` in earlier versions of the draft), instead of "*", as now is reflected in the current working draft. In addition, by having developed a very early implementation of this module, I was able to contribute to clarify some parts of the specification, which behaviour was not very clear.

This thesis also purposes other original proposals that have not yet been accepted to form part of the current working draft, to wit:

- Non-rectangular slots (see pp. 212 and 217)
- A detailed algorithm for computing heights (p. 203)

- Removing the constraint on which properties can be applied to slots (p. 219)
- Allowing elements to positioned into any template, regardless of whether they are children or not (p. 200)
- Using percentages for the specifying the width of columns (p. 216)

This template mechanism fulfil the aforementioned requirements for a layout proposal for CSS, to wit:

- Templates allow to define the layout in a **explicit** way
- Layout (that is, the visual structure of the document when it is rendered) is **independent on the content order** and the structure of HTML document

ALMCSS: The First Implementation of the CSS3 Template Layout Module

This thesis also presents ALMCSS, which has been for more than three years the only available implementation of the CSS3 Template Layout Module. The first version of this prototype, which has been thoroughly described in *Chapter 11*, was funded by the research project *Extensión del estándar CSS3 que permita la adaptación multidispositivo de contenidos web* (Acebal, Rodríguez, García, Cueva & Labra, 2006), granted by CTIC Foundation. It was first presented in the *World Wide Web Conference* (Bos & Acebal, 2006) and then in a co-authored chapter on *Transcending CSS* (Clarke & Acebal, 2007). After that, it has been used by Clarke and Bos to demonstrate the capabilities of the CSS3 Template Layout Module in numerous seminars, workshops, and conferences all around the world.

It is a JavaScript prototype that allows any designer to use the new template mechanism in any web page, simply by including a reference to a JavaScript file in the HTML document: the prototype adds an upper layer over the browser layout engine that is able to understand (although it is true that with certain bugs and limitations) the properties and values of the new template-based layout mechanism proposed in this thesis, and lay out the page accordingly.

The main achievement of the prototype is that, thanks to be implemented in JavaScript, it works on most current web browsers, which has allowed to *see* the proposed template layout extension in action since the earliest phases of the CSS3 Template Layout Module development. This has been useful not only for us, the W3C CSS Working Group, and specially for Bert and me as editors of the Working Draft, but also for showing the benefits of template layout to web designers.

In addition, it has also proved that the proposed solution should not be difficult to implement by browser vendors, if the Template Layout Module is eventually accepted to form part of the future CSS3 specification.

A Visual Tool for Generating Templates

In addition to ALMcss, another prototype has been developed for this thesis, consisting on a visual tool for generating templates (Abella, 2009). This has been an unplanned achievement of this thesis: by fulfilling the aforementioned requisite of allowing the layout to defined explicitly, the Template Layout Module makes possible for a visual tool to create layouts generating a CSS code as clean and understandable as it had been coded by hand (and, of course, without altering the HTML document).

I think that it is very significative of how different the approach of the Template Layout Module is with respect traditional CSS layout mechanisms (floats, absolute positioning, negative margins, etcetera) that we have been able to *automatise* the layout creation process in an undergraduate student thesis, something that no commercial tool does. This is not a merit of Abella's work over products like Adobe Dreamweaver or Microsoft Expression Web, of course: it is a merit of Template Layout Module over floats, absolute positioning, and the rest of properties currently used in CSS for layout.

It is simply not possible for a WYSIWYG tool to create layouts using drag and drop but by means of absolute positioning, which leads to the problems that have also been thoroughly discussed in this dissertation. Other tools have appeared, sometimes as extensions to these programs, as is the case of CSS Sculptor (WebAssist,

2009) and CSS Layout Magic (Project Seven, n.d.) extensions for Dreamweaver, but they are not more than a set of predefined templates, with a few customisable options. Conversely, the prototype developed for this thesis takes advantage of the ease of Template Layout Module for defining layout and makes possible to define a layout made of any number of rows, columns, and slots. Furthermore, it allows to arrange the content into the slots by drag and drop.

Of course, it is not more than a prototype, but I think that it is a good proof of concept of the tools that may appear in a near future if the Template Layout Module becomes a W3C Recommendation.

Publications and other Stuff

This section enumerates the publications originated until now by this thesis, research projects, received awards, and related undergraduate theses carried out under my supervision.

Publications

This thesis has led to the following publications:

- Co-editor of *W3C CSS3 Template Layout Module* (accepted on 19th August 2009)
- Bos, B., & Acebal, C. (2006, May). CSS Advanced Layout. In B. Bos (Chair), Style and layout: Key successes to create interoperable web pages. Session conducted at the 15th International World Wide Web Conference (WWW'06), Edinburgh, Scotland. Slides available at `http://www.w3.org/2006/05/w3c-track.html`
- Clarke, A., & Acebal, C. (2007). Advanced Layout. In *Transcending CSS: The fine art of web design* (pp. 345–357) Berkeley, CA: New Riders.

Research Projects

- *Extensión del estándar CSS3 que permita la adaptación multidispositivo de contenidos web*. Research project funded by CTIC Foundation (project code FUO-EM-115-05) (Acebal, Rodríguez, García, Cueva & Labra, 2006)

Awards

The aforementioned research project, and the prototype that was built for it, was the award-winner research project in the *I Premios Sociedad Información en Asturias* (First Asturias Information Society Awards), promoted by the regional government of the Principality of Asturias.

Students' Works

The following students' undergraduate theses have been supervised by me as part of the research conducted for this thesis:

- Cabal, E. J. (2006). *Adaptación de los estándares web para dispositivos móviles*. (Undergraduate thesis, Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo (EUITIO), University of Oviedo).
- Rodríguez, M. (2007). *Implementación de un prototipo visualizador multinavegador para dar soporte al modelo de maquetación avanzado CSS3*. (Undergraduate thesis, Escuela Universitaria de Ingeniería Técnica en Informática de Gijón (EUITIG), University of Oviedo).
- Cabal, E. J. (2009). *Extensión en JavaScript que incorpora el Advanced Layout Module de CSS3 a los navegadores web actuales*. (Master's thesis, Escuela Politécnica Superior de Ingenieros de Gijón (EPSIG), University of Oviedo).
- Abella, P. (2009). *Generador de plantillas del módulo Template Layout de CSS3*. (Undergraduate thesis, Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo (EUITIO), University of Oviedo).

Further Research

More Layout Improvements

The Template Layout Module does not pretend to be the solution to every imaginable layout problem, neither in its current form, nor even assuming that all my proposals were eventually accepted. Layout, as it has been reviewed on *Chapter 2*, is a not so easy to define

concept. It is the perceived result of a complex interaction among many factors, such as hierarchy, alignment, typography, colours, shapes, etcetera. Thus, it is clear that a diagonal text influences the layout differently than an horizontal or vertical one, for example. The Template Layout Module do not address that kind of transformations, the same way that it do not deal with rounded boxes or gradients. All of that may —or may not— need to be added to CSS to achieve the goal pursued by this thesis: a true separation between presentation and content.

For such research, I suggest to take as an inspiration not only other layout languages but, above all, *desktop publishing tools*. It is true that the web is a very different and more complex scenario than printed media, like magazines, newspapers, books, brochures, etcetera, because web design will always mean *device independency*, and there are many factors that are out of the designer's control. But if graphic designers are able to do any design using these tools, we should ask themselves which of their features are missing in Cascading Style Sheets, and whether they can —and it has sense— be implemented or not.

Some of such features that in my opinion are worth to be considered are, for example:

- Allowing the content to flow between whatever two elements
- Allowing the content to fit their containing box
- Substituting rectangular boxes by any shape containing blocks (for example, defined with SVG)

In addition, some others approaches to CSS and, more specifically, to how layout can be specified in CSS are possible (for example, adding *constraints* to the language).

CSS Debuggers

Today, CSS development is much a trial and error process. Few people fully understand the specification. Having a tool that allowed us to debug step by step a style sheet would be an invaluable help, both for beginners and experienced users, who thus would be able to find out, for instance, why the browser insists on keeping that

separation between two elements when a zero margin is being specified to both elements, or why that float is being positioned *there* and not *here* as the author pretended.

One of the causes of more frustration when formatting with CSS is the lack of debugging tools. Some tools, like Web Developer Toolbar (Pederick, 2009), or Firebug ("Firebug", 2010) exist from several years ago, and others have recently appeared: Opera Dragonfly (Mills, 2008), Safari Web Inspector (Apple, 2010), and IE8 Developer Tools (Microsoft, 2009d). Although all of them are very useful tools for web designers, they are *inspectors*: there is still nothing like a true debugger. It is also curious that all of those tools are extensions of web browsers: there is no reason for such integration. I predict that some professional CSS debugger, probably developed as a stand alone application, will appear sooner or later.

Applying Design Patterns and other Object Oriented Best Practices to Layout Engine Construction

That the development of a layout engine, as it is a web browser, is a very complex process is something that nobody has ever disputed. However, from the review of open source browsers Firefox and WebKit I conclude that it is not as complex as it seems when one looks how their layout engines are designed and implemented, and that they could benefit from a more elegant object oriented design by means of design patterns, refactoring, automated unit testing, and other best practices in software design. Of course, a more detailed review of their code would be needed to be able to sustain that affirmation, and that is the reason why it is only pointed here as a suggestion of further research, since it has not been demonstrated in the body of the dissertation (and it is something that it is out of the scope of this thesis indeed).

13

Conclusiones e investigación futura

El último capítulo de esta tesis resume las que en mi opinión son las sus principales aportaciones al problema de la maquetación en la web. Primeramente, y como introducción al capítulo, se ofrece un repaso sumarísimo de la tesis completa, antes de pasar a las aportaciones en sí. Por último, se enumeran y describen brevemente aquellas áreas que han quedado abiertas y, a mi juicio, son merecedoras de más investigación.

Repaso

Esta disertación comenzaba estableciendo la hipótesis de que CSS no es un lenguaje de maquetación. A pesar de la asunción comúnmente aceptada de que Cascading Style Sheets permite la separación de presentación y contenido, en mi opinión dicha separación no es posible en la actualidad. Ni los elementos flotantes ni el posicionamiento absoluto, o el resto de propiedades del lenguaje que tratan con el modelo de cajas y de formato visual, pueden considerarse verdaderos mecanismos de maquetación. Como resultado, la maquetación tiende a ser dependiente, en mayor o menor medida, del marcado del documento, lo que invalida la prometida separación entre presentación y contenido.

Además, maquetar con CSS dista mucho de ser una tarea fácil: los *floats*, el posicionamiento, tanto absoluto como relativo, y los márgenes, son mecanismos de bajo nivel, que operan sobre los elementos individuales. De este modo, la estructura visual final de la página no es sino el resultado de las complejas interacciones que acontecen entre las citadas propiedades aplicadas a cada elemento de la página, de acuerdo con las reglas que se definen en CSS 2.1, reglas éstas que pueden llegar a ser, en ocasiones, muy complejas.

Esta tesis ha abordado el problema de la maquetación en la web siguiendo un enfoque totalmente distinto a como se ha venido haciendo hasta ahora en CSS: en vez de especificar la estructura visual de la página manipulando las propiedades de cada elemento individual, ésta se define ahora de manera **explícita**, a un nivel de abstracción más alto, que utiliza los mismos conceptos a los que los diseñadores están acostumbrados en el diseño gráfico tradicional, en los medios impresos: filas y columnas, así como los módulos formados por la combinación de varias filas y columnas. Esto representa un cambio drástico en el modo de maquetar con CSS.

Los diseños así creados no sólo son mucho más fáciles de conseguir que con los mecanismos tradicionales de CSS, sino que también son independientes de la estructura del documento: no importa cuál sea la posición de un elemento en el código fuente del docu-

mento, éste podrá ubicarse en cualquier lugar de página, proporcionando así una suerte de **mecanismo de reordenación del contenido**, que esta tesis ha demostrado que es una característica esencial para obtener una verdadera separación de presentación y contenido o, más concretamente, entre la estructura del documento y su jerarquía visual.

Principales aportaciones

En este apartado se resumen cuáles son las principales aportaciones de esta tesis.

CSS no es un lenguaje de maquetación

La primera aportación es la propia hipótesis de la tesis, expresada en cualquiera de sus múltiples variantes: "CSS no es un lenguaje de maquetación", "la separación de presentación y contenido no es posible actualmente en la web", "las hojas de estilo en cascada carecen de verdaderos mecanismos de maquetación" "no es posible rediseñar con CSS"… son todas ellas afirmaciones arriesgadas, que van en contra de la verdad establecida. Si bien es cierto que algunos autores ya se habían *rebelado* contra está asunción, aún es una afirmación que podríamos calificar de *políticamente incorrecta*. No obstante, en mi opinión ha sido suficientemente probada en esta tesis y, aunque sólo fuera por ello —es decir, si verdaderamente he sido capaz de demostrar que son necesarios mecanismos de posicionamiento más avanzados en CSS, ya sean los de la solución propuesta de esta tesis u otros—, creo que esta tesis ya habría valido la pena.

CSS es un lenguaje de bajo nivel

Otra aportación que considero particularmente interesante es la comparación hecha en el capítulo 8 (p. 190) entre Cascading Style Sheets y los lenguajes de programación, atendiendo al nivel de abstracción. Sostengo que maquetar con CSS es como pretender crear una aplicación compleja en código ensamblador, y que es preciso dotar al lenguaje de hojas de estilo con mecanismos de posicionamiento más avanzados.

Maquetación implícita frente a maquetación explícita

Otra aportación original de esta tesis es la distinción hecha entre maquetación implícita y explícita. Por maquetación *implícita* entiendo el modo en que ésta se define hoy día en CSS, como ya se ha repetido en el repaso introductorio de este capítulo. No sólo es difícil crear la maquetación de una página con los mecanismos actuales de maquetación de CSS: es incluso más complicado tratar de averiguar, a partir de un documento HTML y su hoja de estilo asociada, qué maquetación está siendo aplicada. Por el contrario, en esta tesis defiendo que una es necesario un mecanismo para definir la maquetación de manera *explícita*, donde el autor pueda decir que una página tiene, por ejemplo, tres columnas, una cabecera y un pie de página, y que la columna del medio es líquida mientras que las de los lados tienen un ancho de 13 em y 225 píxeles, respectivamente, y que a continuación pueda ubicar los elementos a su antojo simplemente indicando qué elementos van en cuáles de las áreas definidas previamente.

Cascading Style Sheets requiere distinguir la maquetación del resto de aspectos de presentación

Ésta es otra distinción poco convencional hecha por esta tesis: en ella distingo la "maquetación" del resto de información estilística de la página. Normalmente dichos conceptos se encuentran englobados en el término "presentación" pero, mientras que las hojas de estilo han funcionado razonablemente bien para muchos de esos otros aspectos, realizando una gran labor en lo que se refiere a sacar la mayoría de los aspectos de presentación fuera del HTML, no se puede decir lo mismo de la maquetación de la página. Gracias a diferenciar esta última como un subconjunto de lo que comúnmente se entiende por *presentación* he hecho posible centrarme en ese problema en concreto, tanto desde un punto de vista meramente conceptual, como a la hora de proponer una solución.

Requisitos para un sistema de maquetación en CSS

La primera parte de la memoria de la tesis se ha centrado en analizar por qué las hojas de estilo no son adecuadas para maquetar, y lo ha

demostrado a través de unos pocos casos de estudio seleccionados. Los problemas identificados se describieron en el capítulo 8, donde también se definieron los dos principales requisitos que cualquier propuesta de adición a CSS debería satisfacer. Dichos requisitos son los siguientes:

Debe proporcionar un mecanismo de reordenación del contenido

Es decir, debe ser completamente independiente del orden y estructura del código fuente del documento, de modo que cualquier elemento pueda ser colocado en cualquier posición de la página, no importa dónde se haya definido éste en el HTML. Naturalmente, esto debe lograrse cumpliendo a la vez los requisitos arquitectónicos de la web, es decir, la capacidad de que la página pueda verse en cualquier dispositivo, sean cuales sean sus características, tales como el tamaño de la pantalla o las fuentes instaladas.

La maquetación debe ser explícita

Como ya se ha explicado, uno de los principales problemas derivados de cómo se especifica la maquetación de una página hoy día en CSS es que ésta está implícitamente definida como resultado de todas las propiedades de bajo nivel aplicadas a los elementos individuales. En vez de eso, la maquetación debería ser explícita: esta página tendrá esta y aquella regiones, cada una de las cuales tendrá estas dimensiones —que deben poder ser definidas en relación con las de otros elementos de la página—, y aquí irán estas partes del contenido, mientras que estas otras irán en aquella región.

Se propone un innovador sistema de maquetación

Ésta es probablemente la aportación más obvia de esta tesis: el Template Layout Module propuesto en el capítulo 9 como solución al problema de la maquetación en la web. Sin embargo, como ya se dijo en aquel capítulo, no puedo pretender apropiarme del trabajo de otros: siendo ésta una solución desarrollada en el seno del W3C CSS Working Group (W3C CSS-WG), dicha propuesta es, por definición, el resultado del esfuerzo de muchas otras personas. Especialmente de Bert Bos, uno de los directores de esta tesis, quien ya había

definido la sintaxis y comportamiento básicos del módulo (por aquel entonces denominado Advanced Layout) cuando comencé esta tesis y me uní al W3C CSS Working Group.

Una de mis contribuciones a la versión actual del CSS3 Template Layout Module es la capacidad de aplicar estilo a los propios elementos (el pseudo-elemento `::slot(x)`). Otras son probablemente menos visibles, pero aun así importantes. Por ejemplo, en su día sugerí que la altura predeterminada de las filas debería tener el valor `auto` (entonces denominado `intrinsic`), en vez de asterisco ("*"), propuesta aceptada y que aparece ahora recogida en la versión actual del borrador de la especificación. Además, durante el desarrollo de nuestro prototipo, ALMCSS, pude contribuir a clarificar algunas partes de la especificación cuyo significado no estaba claro.

En la tesis también se proponen otras adiciones o cambios al módulo que aún no han sido aceptadas para formar parte de la especificación, a saber:

- Regiones no rectangulares (ver pp. 212 y 217)
- Un algoritmo detallado para el cálculo de la altura (p. 203)
- Eliminar las restricciones acerca de qué propiedades pueden ser aplicadas a los slots (p. 219)
- Permitir que los elementos puedan posicionarse en cualquier plantilla, independientemente de si son descendientes de ella o no (p. 200)
- Usar porcentajes para especificar el ancho de las columnas (p. 216)

El mecanismo basado en plantillas que se propone en esta tesis satisface los requisitos previamente mencionados para un sistema de maquetación en CSS, a saber:

- Las plantillas permiten definir la maquetación de forma **explícita**
- La maquetación (esto es, la estructura visual del documento) es **independiente del orden del contenido** y de la estructura del documento HTML.

ALMcss: La primera implementación del CSS3 Template Layout Module

Esta tesis también presenta ALMcss, la que ha sido durante más de tres años la única implementación disponible del CSS3 Template Layout Module. La primera versión del prototipo, el cual ha sido concienzudamente descrito en el capítulo 11, fue financiada por el proyecto de investigación *Extensión del estándar CSS3 que permita la adaptación multidispositivo de contenidos web* (Acebal, Rodríguez, García, Cueva & Labra, 2006), concedido por la Fundación CTIC. El prototipo se presentó por vez primera en la *World Wide Web Conference* (Bos & Acebal, 2006) y más tarde en un capítulo del libro *Transcending CSS* (Clarke & Acebal, 2007), en el que tuve el honor de participar como coautor de dicho capítulo. Después, el prototipo ha sido usado por Ckarke y por Bos para demostrar las capacidades del CSS3 Template Layout Module en numerosos seminarios, talleres prácticos y congresos de todo el mundo.

Se trata de un prototipo en JavaScript que permite que cualquier diseño utilice las nuevas capacidades de maquetación en cualquier página web, con tan sólo incluir una referencia al fichero JavaScript en el documento HTML: el prototipo añade una capa extra el motor de visualización del navegador que es capaz de entender (si bien es cierto que, en su versión actual, con ciertas limitaciones y errores) las propiedades y valores de la solución propuesta, y visualiza la página correctamente.

El principal logro del prototipo es que, al estar implementado en JavaScript, funciona en la mayoría de los navegadores actuales, lo que ha permitido *ver* la solución propuesta en acción ya desde las primeras fases de desarrollo del módulo. Esto ha resultado útil no sólo para nosotros, el W3C CSS Working Group, y en particular Bert y yo como editores del Working Draft, sino también para mostrar sus ventajas a los diseñadores web.

Por último, el prototipo ha permitido probar que la solución propuesta no debería ser difícil de implementar por los fabricantes de navegadores, en caso de que el Template Layout Module sea finalmente aceptado para formar parte de la futura especificación de CSS3.

Una herramienta visual para generar plantillas

Además de ALMCSS, se ha desarrollado otro prototipo para esta tesis: una herramienta visual para la generación de plantillas (Abella, 2009). Éste ha sido, de hecho, un logro añadido de esta tesis, no previsto inicialmente: gracias a satisfacer el requisito previamente mencionado de permitir que la maquetación se defina de forma explícita, el Template Layout Module posibilita también el desarrollo de herramientas visuales para crear la maquetación de la página, generando un código CSS tan limpio y comprensible como el que hubiera podido codificarse a mano (y, por supuesto, sin alterar el documento HTML).

Pienso que es bastante significativo del enfoque tan diferente que el Template Layout Module sigue con respecto a los mecanismos tradicionales de CSS (floats, posicionamiento absoluto, márgenes negativos, etcétera) el hecho de que se haya podido *automatizar* el proceso de creación de la maquetación en un proyecto fin de carrera desarrollado por un estudiante, algo que ninguna herramienta comercial es capaz de hacer hoy día. Esto no significa, naturalmente, que el trabajo de Abella mejore a productos como Adobe Dreamweaver o Microsoft Expression Web, sino que representa un mérito del Template Layout Module sobre los floats, posicionamiento absoluto, y el resto de propiedades actuales de CSS para maquetar.

Simplemente es imposible para una herramienta WYSIWYG crear la maquetación de una página de forma gráfica, arrastrando y soltando elementos, si no es mediante posicionamiento absoluto, lo que conlleva los problemas que han sido profusamente descritos en esta tesis. Si bien es cierto que han aparecido ciertas herramientas que abordan este problema, a veces como extensiones de esos mismo programas mencionados en el párrafo anterior, como es el caso de las extensiones para Dreamweaver CSS Sculptor (WebAssist, 2009) y CSS Layout Magic (Project Seven, n.d.) éstas no son más que una serie de plantillas predefinidas, con más o menos opciones configurables. Por el contrario, el prototipo desarrollado para la tesis se aprovecha de las facilidades del Template Layout Module para maquetar y permite crear un diseño de cualquier número de filas, columnas, y *slots*. Más aún, permite ubicar el contenido en dichos

slots simplemente arrastrando y soltando los elementos deseados en ellos.

Naturalmente, no se trata más que de un prototipo, pero pienso que constituye una prueba de concepto de las herramientas que aparecerán en un futuro próximo si el Template Layout Module se convierte en una especificación oficial del W3C.

Publicaciones y otros logros

En esta sección se enumeran las publicaciones a las que ha dado lugar hasta ahora esta tesis, los proyectos de investigación obtenidos, así como los premios recibidos y los proyectos fin de carrera involucrados en su realización.

Publicaciones

La tesis ha dado lugar a las siguientes publicaciones:

- Fui nombrado coeditor del *W3C CSS3 Template Layout Module* (el 19 de agosto de 2009)
- Bos, B., & Acebal, C. (2006, May). CSS Advanced Layout. In B. Bos (Chair), Style and layout: Key successes to create interoperable web pages. Session conducted at the 15th International World Wide Web Conference (WWW'06), Edinburgh, Scotland. Slides available at `http://www.w3.org/2006/05/w3c-track.html`
- Clarke, A., & Acebal, C. (2007). Advanced Layout. In *Transcending CSS: The fine art of web design* (pp. 345–357) Berkeley, CA: New Riders.

Proyectos de investigación

- *Extensión del estándar CSS3 que permita la adaptación multidispositivo de contenidos web*. Proyecto de investigación otorgado por la Fundación CTIC (código de proyecto FUO-EM-115-05) (Acebal, Rodríguez, García, Cueva & Labra, 2006)

Premios

Dicho proyecto de investigación, y el prototipo desarrollado para él, resultó ganador del primer premio, en el apartado de investigación,

de los *I Premios Sociedad Información en Asturias*, concedido por el gobierno regional del Principado de Asturias.

Proyectos fin de carrera

Los siguientes proyectos fin de carrera dirigidos por mí forman parte de la investigación realizada para esta tesis:

- Cabal, E. J. (2006). *Adaptación de los estándares web para dispositivos móviles*. (Undergraduate thesis, Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo (EUITIO), University of Oviedo).
- Rodríguez, M. (2007). *Implementación de un prototipo visualizador multinavegador para dar soporte al modelo de maquetación avanzado CSS3*. (Undergraduate thesis, Escuela Universitaria de Ingeniería Técnica en Informática de Gijón (EUITIG), University of Oviedo).
- Cabal, E. J. (2009). *Extensión en JavaScript que incorpora el Advanced Layout Module de CSS3 a los navegadores web actuales*. (Master's thesis, Escuela Politécnica Superior de Ingenieros de Gijón (EPSIG), University of Oviedo).
- Abella, P. (2009). *Generador de plantillas del módulo Template Layout de CSS3*. (Undergraduate thesis, Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo (EUITIO), University of Oviedo).

Investigación futura

Más mejoras de maquetación

El Template Layout Module no pretende ser la solución definitiva a cualquier problema de maquetación inimaginable, ni en su estado actual ni siquiera suponiendo que se aceptaran todas mis propuestas. La disposición visual de un documento, como se estudió en el capítulo 2, no es un concepto sencillo de definir, sino que se trata del resultado de una compleja interacción de muchos otros factores, como la jerarquía, alineación, tipografía, formas y colores, etcétera. Así, es evidente de un texto dispuesto en diagonal ejerce una infl-

uencia sobre la jerarquía visual del documento diferente de si el mismo texto se colocase horizontal o verticalmente, por ejemplo. Sin embargo, el Template Layout Module no aborda este tipo de transformaciones, de la misma manera que no trata con otras cuestiones como bordes redondeados o degradados. Es posible que cuestiones como éstas deban ser añadidas a CSS para lograr el objetivo perseguido por esta tesis, a saber: una separación total entre la presentación y el contenido.

Para dicha investigación posterior, mi sugerencia es que deberían tomarse como inspiración no sólo otros lenguajes de maquetación, sino, sobre todo, las *herramientas de publicación*. Es cierto que la web es un medio muy diferente y representa un escenario más complejo que el de los tradicionales medios impresos, como libros, revistas, periódicos o folletos, dado que el diseño web siempre llevará asociada la *independencia de dispositivos*, y hay muchos factores que, a diferencia del medio impreso, están fuera del control del diseñador. Pero si los diseñadores gráficos han sido capaces de lograr cualquier tipo de diseño en dichos medios, empleando esas herramientas, deberíamos preguntarnos cuáles de sus características no se encuentran en Cascading Style Sheets, si pueden ser implementadas o no, y si tiene sentido hacerlo.

Algunas de dichas características que en mi opinión deberían ser consideradas para su posible inclusión en CSS son, por ejemplo:

- Permitir que el contenido fluya entre dos elementos cualesquiera
- Permitir que el contenido pueda adaptarse a su caja contenedora, en vez de únicamente a la inversa como sucede ahora
- Eliminar la restricción de que las cajas sean siempre rectangulares, y que puedan tener cualquier forma (por ejemplo, definiéndolas mediante SVG)

Por último, también son posibles otros planteamientos del problema de la maquetación en CSS (por ejemplo, añadiendo *restricciones* al lenguaje).

Depuradores CSS

Hoy día, el desarrollo con CSS tiene mucho de prueba y error. Pocos desarrolladores conocen y son capaces de comprender la especificación al completo. Si pudiéramos disponer de herramientas que permitiesen depurar una hoja de estilo paso a paso, sería una gran ayuda, tanto para principiantes como para usuarios expertos. De este modo, podríamos averiguar muy fácilmente, por ejemplo, por qué el navegador "se empeña" en dejar esa separación entre dos elementos cuando se ha especificado un margen de cero para ambos, o porque un elemento flotante se coloca *ahí* en vez de *aquí* (donde pretendiese el autor).

Así pues, una de las causas de mayor frustración cuando diseñamos con CSS es la falta de este tipo de herramientas de depuración. Desde hace ya unos cuantos años, existen algunas herramientas, como la Web Developer Toolbar (Pederick, 2009) o Firebug ("Firebug", 2010), y otras han aparecido recientemente: Opera Dragonfly (Mills, 2008), Safari Web Inspector (Apple, 2010) e IE8 Developer Tools (Microsoft, 2009d). Si bien todas ellas son muy útiles para los diseñadores web, no son más que *inspectores*: todavía no existe nada parecido a un verdadero depurador. También resulta curioso que todas esas herramientas sean extensiones de navegadores web: no hay razón alguna para dicha integración. Mi predicción es que algún depurador profesional de CSS, probablemente desarrollado como una aplicación independiente del navegador, aparecerá tarde o temprano.

Aplicar patrones de diseño y otras buenas prácticas orientadas a objetos a la construcción de motores de renderizado

Que el desarrollo de un motor de renderizado, como es un navegador web, es una tarea compleja, es algo que nadie pone en duda. No obstante, del análisis de los navegadores de código abierto Firefox y WebKit desarrollado para el proyecto de investigación que dio lugar al desarrollo del prototipo de esta tesis, se desprende que parte de esa complejidad es debida precisamente a ciertos errores de diseño, y que podrían beneficiarse de un diseño mucho más elegante y, por qué no decirlo, más orientado a objetos, de haberse aplicado

patrones de diseño, factorización de código, pruebas automatizadas y otras buenas prácticas de diseño de software. Por supuesto, sería preciso un análisis más exhaustivo de su código fuente para poder sostener dicha afirmación, y ése es el motivo por el que simplemente se apunta aquí como sugerencia para una futura investigación, ya que no ha sido demostrado en el cuerpo de la tesis (y es algo que, de hecho, está fuera del ámbito de esta tesis).

References

Abella, P. (2009). *Generador de plantillas del módulo Template Layout de CSS3*. (Undergraduate thesis, Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo (EUITIO), University of Oviedo).

Acebal, C., Rodríguez, M., García, M., Cueva, J. M., & Labra, J. E. (2006). *Extensión del estándar CSS3 que permita la adaptación multidispositivo de contenidos web.* (Technical Report). Gijón, Spain: CTIC Foundation.

Acebal, C., F., Cueva, J., M., & Izquierdo, R. (2003). Usabilidad como factor clave en el comercio electrónico. In L. Joyanes, M. González (Eds.) *Libro de actas del II Congreso Internacional de Sociedad de la Información y del Conocimiento (CISIC'03)* (Tomo 2, pp. 305–310). España: McGraw-Hill.

Acebal, C. (2001). *Lenguaje para el desarrollo ágil de software: Un enfoque basado en patrones de diseño* (Technical Report). Trabajo de investigación para la obtención de la Suficiencia Investigadora. Gijón, Spain: Computer Science Department, University of Oviedo.

Acebal, C., F., Izquierdo, R., & Cueva, J. M. (2001). Good design principles in a compiler university course. *ACM SIGPLAN Notices*, *37*(4), 62–73. doi:10.1145/510857.510870

Adams, C., Edwards, J., Heilmann, C., Mahemoff, M., Pehlivanian, A., Webb, D., & Willison, S. (2007). *The art & science of JavaScript*. Collingwood, Australia: SitePoint.

Aho, A., V., Sethi, R., Ullman, & J., D., (1990). *Compiladores: Principios, técnicas y herramientas* Wilmington, NC: Addison-Wesley Iberoamericana. Translated from *Compilers: Principles, techniques, and tools* (1986). Reading, MA: Addison-Wesley.

Allsopp, J. (2000 April 17). A Dao of web design. *A List Apart*, 58. Retrieved from http://www.alistapart.com/articles/dao/

Ambrose, G. , & Harris, P. (2008). *Grids*. Laussanne, Switzerland: AVA Publishing.

Ambrose, G. , & Harris, P. (2005). *Layout*. Laussanne, Switzerland: AVA Publishing.

Apple. (2010, January 20) Debugging your website. In *Safari Developer Center*. Retrieved from http://developer.apple.com/safari/library/documentation/AppleApplications/Conceptual/Safari_Developer_Guide/DebuggingYourWebsite/DebuggingYourWebsite.html

Baekdal, T. (2006, October 24). Actual browser sizes [Web log message]. Retrieved from http://www.baekdal.com/reports/actual-browser-sizes/

Baron, D. (2008, November 18). Faster HTML and CSS: Layout engine internals for web developers [Video file]. Google Tech Talks. Retrieved from http://www.youtube.com/watch?v=a2_6bGNZ7bA

Baron, D. (2007, November 14). *More precise definitions of intrinsic widths and table layout*. Retrieved from http://dbaron.org/css/intrinsic/

Baron, D. (2006a, December 14). Mozilla's layout engine [HTML slides]. Retrieved from http://www.mozilla.org/newlayout/doc/layout-2006-12-14/master.xhtml

Baron, D. (2006b, May). Layout algorithm improvements for web user interfaces. In *Browser technology*. Session at the XTech 2006 Conference, Amsterdam, The Netherlands. Retrieved from http://xtech06.usefulinc.com/schedule/paper/146

Baron, D. (2003, January 9). Thursday 2003-01-09 [Web log message]. Retrieved from http://dbaron.org/log/2003-01#l20030109

Bernard, M., Fernandez, M., & Hull, S. (2002, July). The effects of line length on children and adults' online reading performance. *Usability News*, *4*(2). Retrieved from http://psychology.wichita.edu/surl/usabilitynews/42/text_length.asp

Berners-Lee, T. (1999). *Weaving the web: The original design and ultimate destiny of the World Wide Web, by its inventor*. New York, NY: HarperCollins Publishers.

Berners-Lee, T. (2007, March 1). Digital future of the United States: Part I – The future of the World Wide Web. Testimony before the United States House of Representatives (Committee on Energy and Commerce). Retrieved from http://energycommerce.house.gov/index.php?option=com_content&view=article&id=291&catid=32&Itemid=58

Berners-Lee, T. (1993). *Hypertext Markup Language (HTML): A representation of textual information and metaInformation for retrieval and interchange (IETF Internet Draft)*. Retrieved from http://www.w3.org/MarkUp/draft-ietf-iiir-html-01.txt

Berners-Lee, T. (1991). *HTML design constraints*. Retrieved from http://www.w3.org/MarkUp/HTMLConstraints.html

Berry, J. D. (Ed.) (2004). *Contemporary newspaper design*. New York, NY: Mark Batty Publisher.

Bos, B., & Acebal, C. (2006, May). CSS Advanced Layout. In B. Bos (Chair), *Style and layout: Key successes to create interoperable web pages.* Session conducted at the 15th International World Wide Web Conference (WWW'06), Edinburgh, Scotland. Slides retrieved from http://www.w3.org/2006/05/w3c-track.html

Bos, G. (1993). *Rapid user interface development with the script language Gist*. (Doctoral dissertation, Rÿks*universiteit* Groningen). Retrieved from http://www.w3.org/People/Bos/

Bos, B. (2009, April 2). CSS3 Template Layout Module (W3C Working Draft). Retrieved from http://www.w3.org/TR/2009/WD-css3-layout-20090402/

Bos, B. (2007a, August 9). CSS3 Advanced Layout (W3C Working Draft). Retrieved from http://www.w3.org/TR/2007/WD-css3-layout-20070809

Bos, B. (2007b, August 9). CSS basic box model (W3C Working Draft). Retrieved from http://www.w3.org/TR/2007/WD-css3-box-20070809

Bos, B. (2005, December 15). CSS3 Advanced Layout (W3C Working Draft). Retrieved from http://www.w3.org/TR/2005/WD-css3-layout-20051215/

Bos, B., Çelik, T., Hickson, I., & Lie, H. W., (2009, September 8). Cascading Style Sheets Level 2 Revision 1 (W3C Candidate Recommendation). Retrieved from http://www.w3.org/TR/2009/CR-CSS2-20090908/

Bos, B., Lie, H. W., Lilley, C., & Jacobs, I. (1998, May 12). Cascading Style Sheets Level 2 (W3C Recommendation). Retrieved from http://www.w3.org/TR/1998/REC-CSS2-19980512/

Bowman, D. (2003, December 15). On fixed vs. liquid design [Web log message]. Retrieved from http://stopdesign.com/archive/2003/12/15/fixedorliquid.html

Bowman, D. (2004, September 3). Liquid bleach [Web log message]. Retrieved from http://stopdesign.com/archive/2004/09/03/liquid-bleach.html

Braganza, C., Marriott, K., Moulder, P., Wybrow, M., & Dwyer, T. (2009). Scrolling behaviour with single- and multi-column layout. In *Proceedings of the 18th international Conference on World Wide Web (WWW'09)* (pp. 831–840). New York, NY: ACM. doi:10.1145/1526709.1526821

Brahmachari, S. (2000, February 4). Graphics in advertising (part III): The main function of graphic design [Web log message].

Retrieved from http://www.suite101.com/article.cfm/advertising/32969

Bringhurst, R. (2005). *The elements of typographic style* (3rd ed.). Vancouver, BC: Hartley & Marks, Publishers.

Brailsford, D. F. (1988). Electronic publishing and computer science. *Electronic Publishing, 0*(January 1988), 13–21.

Brown, F. C. (1921). *Letters & lettering: A treatise with 200 examples*. Boston: Bates & Guild Company. Also available at http://www.gutenberg.org/files/20590/20590-h/20590-h.htm (Project Gutenberg).

Budd, A. (2004, May 12). An objective look at table based vs. CSS based design [Web log message]. Retrieved from http://www.andybudd.com/archives/2004/05/an_objective_look_at_table_based_vs_css_based_design/

Budd, A. (2003, November 23). No margin for error [Web log message]. Retrieved from http://www.andybudd.com/archives/2003/11/no_margin_for_error/

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture: A system of patterns*. Chichester, England: John Wiley & Sons.

Byous, J. (2003, April). Java Technology: The Early Years. Retrieved from http://java.sun.com/features/1998/05/birthday.html

Cabal, E. J. (2009). *Extensión en JavaScript que incorpora el Advanced Layout Module de CSS3 a los navegadores web actuales*. (Master's thesis, Escuela Politécnica Superior de Ingenieros de Gijón (EPSIG), University of Oviedo).

Cabal, E. J. (2006). *Adaptación de los estándares web para dispositivos móviles*. (Undergraduate thesis, Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo (EUITIO), University of Oviedo).

Caldwell, B., Cooper, M., Reid, L., G., & Vanderheiden, G., (2008, December 8). Techniques for WCAG 2.0 (W3C Note). Retrieved from http://www.w3.org/TR/2008/NOTE-WCAG20-TECHS-20081211/

Cederholm, D. (2008). *Bulletproof web design: Improving flexibility and protecting against worst-case scenarios with XHTML and CSS* (2nd ed.). Berkeley, CA: New Riders.

Cederholm, D. (2004, January 9). Faux columns. *A List Apart*, 167. Retrieved from http://www.alistapart.com/articles/fauxcolumns/

Çelik, T. (2004, September 6). Undoing html.css and using debug scaffolding [Web log message]. Retrieved from http://tantek.com/log/2004/09.html#d06t2354

Century Software (2009). ViewML [Computer software] Retrieved from http://www.pixil.org/

Chen, P., & Harrison, M. A. (1988). Multiple representation document development. *Computer*, *21*(1), 15–31.

Chromatic (2008, April 13). 13 reasons why CSS is superior to tables in website design [Web log message]. Retrieved from http://www.chromaticsites.com/blog/13-reasons-why-css-is-superior-to-tables-in-website-design/

Chisholm, W., Vanderheiden, G., & Jacobs, I., (1999, May 5). Web Content Accessibility Guidelines 1.0 (W3C Recommendation). Retrieved from http://www.w3.org/TR/WCAG10/

Chisholm, W., Vanderheiden, G., & Jacobs, I., (2000, November 6). Techniques for Web Content Accessibility Guidelines 1.0 (W3C Note). Retrieved from http://www.w3.org/TR/WCAG10-TECHS/

Chisholm, W., Vanderheiden, G., & Jacobs, I., (2000, November 6). HTML Techniques for Web Content Accessibility Guidelines 1.0 (W3C Note). Retrieved from http://www.w3.org/TR/WCAG10-HTML-TECHS/

Clark, P. (2008). Content management and the separation of presentation and content. *Technical Communication Quarterly*, *17*(1), 35–60.

Clarke, A., & Acebal, C. (2007). Advanced Layout. In *Transcending CSS: The fine art of web design* (pp. 345–357). Berkeley, CA: New Riders.

Clarke, A. (2007a). *Transcending CSS: The fine art of web design*. Berkeley, CA: New Riders.

Clarke, A., (2007b). Designing for outside the box. In *Web standards creativity: Innovations in web design with XHTML, CSS, and DOM scripting* (pp. 78–107). Berkeley, CA: friendsof.

Coombs, J. H., Renear, A. H., & DeRose, S. J. (1987). Markup systems and the future of scholarly text processing. *Communications of the ACM*, *30*(11), 933–947. doi:10.1145/32206.32209

Crockford, D. (2008). *JavaScript: The good parts*. Sebastopol, CA: O'Reilly Media.

Crockford, D. (n.d.). JavaScript [Several essays and presentations]. Retrieved from http://javascript.crockford.com/

Crockford, D. (2001). JavaScript: The World's most misunderstood programming language. Retrieved from http://javascript.crockford.com/javascript.html

Cullen, K. (2005). *Layout workbook*. Gloucester, MA : Rockport Publishers.

Deveria, A. (2009, April 23). Ready for use: CSS3 Template Layout [Web log message]. Retrieved from http://a.deveria.com/?p=236

Edwards, D. ie7-js (Version 2.1 beta 2). (2010, February). [Computer software] Retrieved from http://code.google.com/p/ie7-js/

Edwards, D. IE7 (Version 0.9 alpha). (2005, August 19). [Computer software] Retrieved from http://dean.edwards.name/weblog/2005/09/ie7-09

Evans, P. (2006). *Exploring publication design*. Clifton Park, NY: Thomson Delmar Learning.

Firebug (Version 1.5.2). (2010, February). [Computer software] Retrieved from http://getfirebug.com/

Fisher, D. (2009, June 2). Exploring Chrome internals [Video file]. In Google I/O 2009. Retrieved from http://www.youtube.com/watch?v=Naol_TPPPL0

Flanagan, D. (2006). *JavaScript: The definitive guide* (5th ed.). Sebastopol, CA: O'Reilly Media.

Fleishman, G. (1999). Styling the web. *Adobe Magazine*, Autumn 1999. Retrieved from http://www.adobe.com/products/adobemag/archive/autm99na.html

The Flying Saucer Project (Release 8). (2009). [Computer software] Retrieved from https://xhtmlrenderer.dev.java.net/

Furman, S., & Isaacs, S. (1997, January 31). Positioning HTML elements with Cascading Style Sheets (W3C Working Draft). Retrieved from http://www.w3.org/TR/WD-positioning-970131

Furuta, R., Quint, V., & André, J. (1988). Interactively editing structured documents. *Electronic Publishing, 1*(1), 19–44.

Gamma, E. , Helm, R., Johnson, R., & Vlissides, J. M. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison Wesley Longman.

Gillenwater, M. Z. (2009). *Flexible web design* Berkeley, CA: New Riders.

Goldfarb, C., F. (1981). A generalized approach to document markup. *ACM SIGPLAN Notices*, *16*(6), 68–73. doi:10.1145/872730.806456

Graham, L. (2008). Gestalt theory in interactive media design. *Journal of Humanities & Social Sciences*, *2*(1), 1–12.

Grier, C. King, S. T., & Wallach, D. S. (2009, May). How I learned to stop worrying and love plugins. Web 2.0 Security and Privacy. Oakland, CA. Retrieved from http://w2spconf.com/2009/

Hicksdesign (2004, May 20). 3D CSS Box Model [Web log message]. Retrieved from http://hicksdesign.co.uk/journal/3d-css-box-model

Harmes, R., & Diaz, D. (2008). *Pro JavaScript design patterns*. Berkeley, CA: Apress.

Holzschlag, M. E. (2005a, December 19). Thinking outside the grid. *A List Apart*, 209. Retrieved from http://www.alistapart.com/articles/outsidethegrid

Holzschlag, M. E. (2005b, April). The more things stay the same, the more they change. *Design In-Flight*, 4, 28–33.

Hurst, N., Li, W., & Marriott, K. (2009). Review of automatic document formatting. In *Proceedings of the 9th ACM Symposium on Document Engineering (DocEng'09)* (pp. 99–108). New York, NY: ACM. doi:10.1145/1600193.1600217

Jacobs, I., & Walsh, N. (2004, December 15). Architecture of the World Wide Web, volume one (W3C Recommendation). Retrieved from http://www.w3.org/TR/2004/REC-webarch-20041215/

Izquierdo, R., Acebal, C. F., & Cueva, J. M. (2002). An object oriented entity relationship model framework (ERM). In A. M. Moreno, R. Lee, N. Juristo, & B. Boehm (Eds.) *3rd ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'02)* (pp. 353–360). Madrid, Spain.

Jobs, S. (2003, November 30). The guts of a new machine. Interview with Walker R. *The New York Times*. Retrieved from

http://www.nytimes.com/2003/11/30/magazine/30IPOD.htm

jQuery JavaScript Library (Version 1.4.2). (2010, January). [Computer software] Retrieved from http://jquery.com/

Keith, J. (2005). *DOM scripting*. Berkeley, CA: friendsof.

Keith, J. (2006). *Pro JavaScript techniques*. Berkeley, CA: Apress.

Keith, J. (2003, December 15). The Long Debate [Web log message]. Retrieved from http://adactio.com/journal/750

King, A. B. (2008). *Website optimization: Speed, search engine & conversion rate secrets*. Sebastopol, CA: O'Reilly Media.

Korpela, J. (1998). Lurching Toward Babel: HTML, CSS, and XML. *Computer, 31*(7), 103–104, 106. doi:10.1109/2.689682

Koch, P. (2005, March 8). Clearing floats [Web log message]. Retrieved from http://www.quirksmode.org/blog/archives/2005/03/clearing_floats.html

Krug, S. (2000). *Don't make me think! A common sense approach to web usability*. Indianapolis, IN: Que.

Levering, R., & Cutler, M., (2006). The portrait of a common HTML web page. In *Proceedings of the 2006 ACM Symposium on Document Engineering (DocEng'06)* (pp. 198–204). New York, NY: ACM. doi:10.1145/1166160.116621

Lie, H. W. (2007, August 28). CSS @ ten: The next big thing. *A List Apart*, 244. Retrieved from http://www.alistapart.com/articles/cssatten

Lie, H. W. (2005). *Cascading Style Sheets*. (Doctoral dissertation, Faculty of Mathematics and Natural Sciences, University of Oslo). Retrieved from http://people.opera.com/howcome/2006/phd/

Lie, H. W., & Bos, B. (2005). *Cascading Style Sheets: Designing for the web* (3rd ed.). Upper Saddle River, NY: Pearson Education.

Lie, H. W., & Saarela, J. (1998). Multipurpose Web publishing using HTML, XML, and CSS. *Communications of the ACM, 42*(10), 95–101. doi:10.1145/317665.317681

Lie, H. W., & Bos, B. (1996, December 17). Cascading Style Sheets Level 1 (W3C Recommendation). Retrieved from http://www.w3.org/TR/2008/REC-CSS1-20080411

Livingstone, D. (n.d.). Interactive CSS Box Model demo [Web log message]. Retrieved from http://www.redmelon.net/tstme/box_model/

Lynch, P. J., & Horton, S. (2009). Visual design principles. In *Web style guide* (3rd ed.), chapter 7: Page design. Retrieved from http://www.webstyleguide.com/wsg3/7-page-design/4-visual-design-principles.html

Lynch, P. J., & Horton, S. (2009). Presenting information architecture. In *Web style guide* (3rd ed.), chapter 3: Information architecture. Retrieved from http://webstyleguide.com/wsg3/3-information-architecture/4-presenting-information.html

Marcotte, E. (2003, December 15). Further Roasting the Chestnut [Web log message]. Retrieved from http://sidesh0w.com/weblog/2003/12/15/further_roasting_the_chestnut/

McWade, J. (2009, May 19). What is graphic design? [Web log message] Retrieved from http://www.mcwade.com/DesignTalk/2009/05/what-is-graphic-design/

Merikallio, B., & Pratt, A. (2003). Why tables for layout is stupid: Problems defined, solutions offered. Retrieved from http://www.hotdesign.com/seybold/

Meyer, E. (2008a, February 12). CSS tools: Reset CSS [Web log message]. Retrieved from http://meyerweb.com/eric/tools/css/reset/

Meyer, E. (2008b, January 15). Resetting again [Web log message]. Retrieved from http://meyerweb.com/eric/thoughts/2008/01/15/resetting-again/

Meyer, E. (2007a). *CSS: The definitive guide* (3rd ed.). Sebastopol, CA: O'Reilly Media.

Meyer, E. (2007b, April 18). Reset reasoning [Web log message]. Retrieved from http://meyerweb.com/eric/thoughts/2007/04/18/reset-reasoning/

Meyer, E. (2007c, April 14). Reworked reset [Web log message]. Retrieved from http://meyerweb.com/eric/thoughts/2007/04/14/reworked-reset/

Meyer, E. (2007d, April 12). Reset styles [Web log message]. Retrieved from http://meyerweb.com/eric/thoughts/2007/04/12/reset-styles/

Meyer, E. (2004a, November 3). Uncollapsing margins [Web log message]. Retrieved from http://complexspiral.com/publications/uncollapsing-margins/

Meyer, E. (2004b, September 3). Sliding faux columns [Web log message]. Retrieved from http://meyerweb.com/eric/thoughts/2004/09/03/sliding-faux-columns/

Meyer, E. (2004c, July 24). Floats don't suck if you use them right [Web log message]. Retrieved from http://meyerweb.com/eric/thoughts/2004/07/17/floats-dont-suck-if-you-use-them-right/

Meyer, E. (2003a, October 15). The incomplete divorce [Web log message]. Retrieved from http://www.meyerweb.com/eric/thoughts/200310.html#t200310015

Meyer, E. (2003b, August 25). Containing floats [Web log message]. Retrieved from http://complexspiral.com/publications/containing-floats/

Microsoft. (2009a). Introduction to Windows Presentation Foundation. In *Windows Presentation Foundation: Getting started*. Retrieved from *MSDN Library* website: http://msdn.microsoft.com/library/aa970268.aspx

Microsoft. (2009b). XAML. In *Windows Presentation Foundation: WPF fundamentals*. Retrieved from *MSDN Library* website: http://msdn.microsoft.com/library/ms747122.aspx

Microsoft. (2009c). The layout system. In *Windows Presentation Foundation: The layout system*). Retrieved from *MSDN Library* website: http://msdn.microsoft.com/library/ms745058.aspx

Microsoft. (2009d). Debugging HTML and CSS with the Developer Tools. Retrieved from *MSDN Library* website: http://msdn.microsoft.com/library/dd565627(VS.85).aspx

Mills, C., B. , & Weldon, L. J. (1987). Reading text from computer screens. *ACM Computing Surveys, 19*(4), 329–357. doi:10.1145/45075.46162

Mozilla. (2009a) Source code directories overview. In *Mozilla Developer Center*. Retrieved from https://developer.mozilla.org/en/Source_code_directories_overview

Mozilla. (2009b) XPCOM. In *Mozilla Developer Center*. Retrieved from https://developer.mozilla.org/en/XPCOM

Mozilla. (2009c) Adding a new style property. In *Mozilla Developer Center*. Retrieved from https://developer.mozilla.org/en/Adding_a_new_style_property

Mozilla. (2009d). XUL Reference. Retrieved from *Mozilla Developer Center* website: https://developer.mozilla.org/en/XUL_Reference

Mozilla. (2007, August 15). The Box Model. In *XUL Tutorial.* Retrieved from *Mozilla Developer Center* website: https://developer.mozilla.org/en/XUL_Tutorial/The_Box_Model

Mullenweg, M. (2003, December 11). Death of Flexible Width Designs [Web log message]. Retrieved from http://ma.tt/2003/12/death-of-flexible-width-designs/

Müller-Brockmann, J. (1971). *Gestaltungsprobleme des grafikers/The graphic artist and his design problems/Les problemes d'un artiste graphique*. Teufen AR, Switzerland: Verlag Arthur Niggli.

Müller-Brockmann, J. (1981). *Grid systems in graphic design/Raster systeme für die visuele gestaltung*. Sulgen, Switzerland: Arthur Niggli.

Nielsen, J. (2000a). *Usabilidad: Diseño de sitios web*. Madrid, Spain: Pearson Educación. Translated from *Designing web usability: The practice of simplicity* (1999). Thousand Oaks, CA: New Riders Publishing.

Nielsen, J. (2009a, July 20). Mobile usability. *Alertbox*. Retrieved from http://www.useit.com/alertbox/mobile-usability.html

Nielsen, J. (2009b, February 17). Mobile web 2009 = desktop web 1998. *Alertbox*. Retrieved from http://www.useit.com/alertbox/mobile-2009.html

Nielsen, J. (2006a, July 31). Screen Resolution and Page Layout. *Alertbox*. Retrieved from http://www.useit.com/alertbox/screen_resolution.html

Nielsen, J. (2006b, April 17). F-shaped pattern for reading web content. *Alertbox*. Retrieved from http://www.useit.com/alertbox/reading_pattern.html

Nielsen, J. (2005, July 11). Scrolling and scrollbars. *Alertbox*. Retrieved from http://www.useit.com/alertbox/20050711.html

Nielsen, J. (2000, October 29). Flash: 99% bad. *Alertbox*. Retrieved from http://www.useit.com/alertbox/20001029.html

Nielsen, J. (1997a, October 1). How users read on the web. *Alertbox*. Retrieved from http://www.useit.com/alertbox/9710a.html

Nielsen, J. (1997b, July 1). Effective use of style sheets. *Alertbox*. Retrieved from http://www.useit.com/alertbox/9707a.html

O'Brien, P. (n.d.). http://pmob.co.uk/

Olsson, T., & O'Brien, P. (2008). *The ultimate CSS reference*. Collingwood, Australia: SitePoint.

Mills, C. Introduction to Opera Dragonfly (2008, May 6). [Web log message]. Retrieved from http://dev.opera.com/articles/view/introduction-to-opera-dragonfly/

van Ossenbruggen, J., & Hardman, L. (2002, May). Smart style on the semantic web. In *Semantic Web Workshop*. Workshop at the 11th International World Wide Web Conference (WWW'02), Hawaii, USA. Retrieved from http://semanticweb2002.aifb.uni-karlsruhe.de/index.htm

van Ossenbruggen, J. (2001). *Processing structured hypermedia: A matter of style*. (Doctoral dissertation, Vrije Universiteit). Retrieved from http://homepages.cwi.nl/~jrvosse/thesis/

Pederick, C. (2009, June). Web Developer Toolbar (Version 1.1.8). [Computer software] Retrieved from http://chrispederick.com/work/web-developer/

Poggenpohl, S. H. (Ed.). (1993). What is graphic design? In *AIGA Career Guide*. Retrieved from http://www.aiga.org/content.cfm/guide-whatisgraphicdesign

Prototype JavaScript Framework (Version 1.6.1). (2009, September). [Computer software] Retrieved from http://www.prototypejs.org/

Project Seven (n.d.). CSS Layout Magic. [Computer software] Retrieved from http://www.projectseven.com/products/templates/pagepacks/cssmagic/index.htm

Raggett, D., Le Hors, A., & Jacobs, I. (1999, December 24). HTML 4.01 Specification (W3C Recommendation). Retrieved from http://www.w3.org/TR/html4/

Raggett, D., Le Hors, A., & Jacobs, I., (1998, April 24). HTML 4.0 Specification (W3C Recommendation, revised). Retrieved from http://www.w3.org/TR/1998/REC-html40-19980424/

Raggett, D., Le Hors, A., & Jacobs, I., (1997, December 18). HTML 4.0 Specification (W3C Recommendation). Retrieved from http://www.w3.org/TR/REC-html40-971218/

Raggett, D. (1997, January 14). HTML 3.2 Reference Specification (W3C Recommendation). Retrieved from http://www.w3.org/TR/REC-html32

Ragget, D. (1996, May). *HTML) tables (IETF RFC 1942)*. Retrieved from http://www.ietf.org/rfc/rfc1942.txt

Reid, B. K. (1981). *Scribe: a document specification language and its compiler*. (Doctoral dissertation, Carnegie Mellon University).

Resig, J. (2009). The DOM Is a Mess [Video file]. In YUI Theather. Retrieved from http://developer.yahoo.com/yui/theater/

Roberts, L. (2007). *Grids: Creative solutions for graphic designers*. Hoboken, NY: John Wiley & Sons.

Robinson, A. (2005, October 21). *In search of the One True Layout*. Retrieved from http://www.positioniseverything.net/articles/onetruelayout/

Rodríguez, M. (2007). *Implementación de un prototipo visualizador multinavegador para dar soporte al modelo de maquetación avanzado CSS3*. (Undergraduate thesis, Escuela Universitaria de Ingeniería Técnica en Informática de Gijón (EUITIG), University of Oviedo).

Rutter, R. (2003, December 15). Fixed Width [Web log message]. Retrieved from http://www.clagnut.com/blog/267/

Samara, T. (2007). *Design elements: A graphic style manual*. Beberly, MA: Rockport Publishers.

Sambells, J., & Gustafson, A., (2007). *Advanced DOM scripting*. Berkeley, CA: friendsof.

Santa Maria, J. (2005, August 24). A List (taken) Apart. Interview with Clarke A. *And all that Malarkey*. Retrieved from http://www.stuffandnonsense.co.uk/archives/a_list_taken_apart.html

Savarese, C. (2005, September 26). Introducing the CSS3 multi-column module. *A List Apart*, 204. Retrieved from http://www.alistapart.com/articles/css3multicolumn/

Shaikh, A., D. (2005, July). The effects of line length on reading online news. *Usability News*, *7*(2). Retrieved from http://www.surl.org/usabilitynews/72/LineLength.asp

Shea, D. (2003). CSS Zen Garden. Retrieved from http://www.csszengarden.com/

Sol, E. (2008, June 17). Faux absolute positioning. *A List Apart*, 261. Retrieved from http://www.alistapart.com/articles/fauxabsolutepositioning/

Stain, B. (2000, November 17). Separation anxiety: The myth of the separation of style from content. *A List Apart*, 89. Retrieved from http://www.alistapart.com/articles/

Sun (2008). LayoutManager. Retrieved from *Java Platform SE 6 API Documentation* website: http://java.sun.com/javase/6/docs/api/java/awt/LayoutManager.html

Tanaka, S. (2009, February 10). How much longer will we design for 1024? [Web log message]. Retrieved from http://www.sohtanaka.com/web-design/how-much-longer-will-we-design-for-1024/

Tondreau, B. (2009). *Layout essentials: 100 design principles for using grids*. Beberly, MA: Rockport Publishers.

Walker, A. (2005, February 26). Simple Clearing of Floats [Web log message]. Retrieved from http://www.sitepoint.com/blogs/2005/02/26/simple-clearing-of-floats/

Walrath, K., Campione, M., Huml, A., & Zakhour, S. (2004). *The JFC Swing tutorial: A guide to constructing GUIs*. (2nd ed.). Redwood City, CA: Addison Wesley Longman. Also available at http://java.sun.com/docs/books/tutorial/uiswing/index.html

WebAssist (2009, September). CSS Sculptor (Version 3). [Computer software] Retrieved from http://www.webassist.com/dreamweaver-extensions/css-sculptor/

WebKit (Nightly Build r43663) [Computer software] (2009, May 13). Retrieved from http://webkit.org/

Website Optimization (2008, April 28). Average web page size triples since 2003 [Web log message]. Retrieved from http://www.websiteoptimization.com/speed/tweak/average-web-page/

Weinschenk, S. (2003, February). Reading text online. *UI Design Newsletter*. Retrieved from http://www.humanfactors.com/downloads/feb03.asp

Weychert, R. (2007). Bridging the type divide: Classic typography with CSS. In *Web standards creativity: Innovations in web design with XHTML, CSS, and DOM scripting* (pp. 156–180) Berkeley, CA: friendsof.

Whitespace (2003, December 12). Death of Liquid Layouts? [Web log message]. Retrieved from http://web.archive.org/web/20031230025612/http://www.9rules.com/whitespace/design/death_of_liquid_layouts.php

Wilkinson, M. J. (2009, May 7). The line length misconception [Web log message]. Retrieved from http://www.viget.com/advance/the-line-length-misconception/

"X-Smiles" (Release 1.2) [Computer software] (2008). Retrieved from http://www.xsmiles.org/

Yahoo! (2009, September). YUI Library (Version 3) [Computer software] Retrieved from http://developer.yahoo.com/yui/

Yahoo! (2009, February). YUI Library: Reset (Version 2.8.0) [Computer software] Retrieved from http://developer.yahoo.com/yui/reset

Yank, K., & Adams, C. (2007). *Simply JavaScript*. Collingwood, Australia: SitePoint.

Zakas, N. C. (2005). *Professional JavaScript for web developers*. Indianapolis, IN: Wiley Publishing.

Zeldman, J., & Marcotte, E. (2010). *Designing with web standards* (3rd ed.) Berkeley, CA: New Riders.

Zeldman, J. (2003). *Designing with web standards*. Indianapolis, IN: New Riders.

Zeldman, J. (2001, February 16). From table hacks to CSS layout: A web designer's journey. *A List Apart*, 99. Retrieved from http://www.alistapart.com/articles/journey/