KTH Computer Science
and Communication

Modular responsive web design

Allowing responsive web modules to respond to custom criterias instead of only viewport size by implementing *element queries*

LUCAS WIENER
lwiener@kth.se

Master's Thesis task specification v1.1

Supervisors at EVRY AB: Tomas Ekholm & Stefan Sennerö
Supervisor at CSC: Philipp Haller
Examiner: Mads Dam

February 2015

Contents

Chapter 1

Task specification

1.1 Problem definition

By using CSS *media queries* developers can specify different rules for different viewport sizes. This is fundamental to create responsive web applications. If developers want to build modular applications by composing the application by smaller components (elements, scripts, styles, etc.) the media queries are no longer applicable. Modular components should be able to react and change style depending on the given size that the component has been given by the application, not the viewport size. The problem can be formulated as: *Elements can not specify conditional style rules depending on their own size, or the size of any other element.* See the included document in section A.2 of the appendix for a more practical problem formulation.

W3C[1] has unofficially stated that such feature would be infeasible to implement. Some problems with implementing element queries in CSS are:

- **Circularity:** The styling of elements depend on many factors (theoretically on all other elements in the layout tree). If elements can apply styles by criterias of other elements, it will be possible to create infinite loops of styling. The simplest example of this would be an element to set its width to 200 pixels if it is under 100 pixels wide. If the element is under 100 pixels wide, the new style will be applied to the element which would make the width of the element 200 pixels. If this element would have another rule that set its width to 50 pixels if it is wider than 150 pixels, there is an infinite loop of styling. Problems like this can probably be catched during CSS parsing, but there are so many combinations of style properties that could result in similar loops that it will add a lot of complexity to the language, both for implementers and users. See the included document in section A.3 of the appendix for more examples and deeper discussions about cyclic rules.
- **Performance:** Rendering engines typically perform selector matching and layout computations in parallel to achieve good performance. If element queries would

[1] World Wide Web Consortium (abbreviated W3C) is the main international standards organization for the World Wide Web.

be implemented, the rendering engines would need to first compute the layout of all elements in order to decide which selectors would conform to the element query conditions and then do a new layout computation, and so on until a stable state has been reached. Far worse, since selectors now depend on layout style, this cannot be done in parallel which impacts performance heavily.

Because of the problems, it is stated that such feature will not be implemented in the near future. So it is now up to the developers to implement this feature as a third-party solution. Efforts have been made by big players to create a robust implementation, with moderate success. Since all implementations have shortcomings, there is still no de facto solution that developers use and the problem remains unsolved.

1.2 Objective

The main objective of my master's thesis is to develop a third-party implementation of element queries (or equivalent to solve the problem of modular responsive elements). To do this, I will need to research and understand all existing attempts and analyze the advantages and shortcomings of each approach. I will also need to be aware of the premises, such as browser limitations and specifications that need to be conformed. In addition, I will research the problems of implementing element queries natively to get a deeper understanding of how an official API would look like. There are many challenges along the way that will need to be researched and worked around. Examples of such subproblems that would need to be investigated are:

- How should circularity be handled? Should it be detected at runtime or parsetime, and what should happen on detection?
- How can one listen to element dimension changes without any native support?
- How can a custom API be crafted that will enable element queries and still conform to the CSS specification?
- If a custom API is developed, how would one make third-party modules (that uses media queries) work without demanding a rewrite of all third-party modules?

The scientific question to be answered is if it is possible to solve the problem without extending the current web standard. The hypothesis is that the problem can be solved in a reliable and performant way by crafting a third-party implementation. A reliable implementation should also enable existing responsive components to react to a specified criteria (parent container size for example) with no modifications to the components. The goal of the thesis should be considered fulfilled if a solution was successfully implemented or described, or if the problems hindering a solution are thoroughly documented.

1.3 Significance

Web developers are today limited to writing big applications in an entagled mess. In almost all other programming environments it is possible and encouraged to write applications in smaller parts (or modules). By creating modules that can be used in any context with well defined responsibilities and dependencies, developing applications is reduced to the task of simply configuring modules (to some extent) to work together which forms a bigger application. It is today possible to write the web client logic in a modular way in JavaScript. The desire of writing modular code can be shown by the popularity of frameworks that helps dividing up the client code into modules. The ever so popular frameworks Angular, Backbone, Ember, Web Components, requirejs, Browserify, Polymer, React and many more all have in common that they embrace coding modular components. Many of these frameworks also help with dividing the HTML up into modules, creating small packages of style, markup and code. One of the biggest issues keeping the modules from being truly modular is that they cannot adapt to given sizes. This makes the modules either force the client to style them properly depending on viewport size, or not being responsive. Both options are undesirable for developing larger applications. A third option would be to make the modules context aware and style themselves according to the viewport, which defeats the purpose of modules (making them not reusable).

The last couple of years a lot of articles have been written about the problem and how badly we need element queries. As already stated, third-party implementation efforts have been made by small and big players, with moderate success. W3C keep getting requests and questions about it, but the answer seems to lean towards no. An organization called Responsive Issues Community Group (abbreviated RICG) have started an initial planning regarding element queries. However, things are moving slow and a draft about element queries use cases are still being made.

Solving this problem would be a big advancement to web development, enabling developers to create truly modular components. By studying the problem, identifying approaches and providing a third-party solution the community can take a step closer to solve the problem. If the hypothesis holds, developers will be able to use element queries in the near future, while waiting for W3C to make their verdict. The outcome of this thesis can also be helpful for W3C and others to get an overview of the problem and possibly get ideas how subproblems can be handled.

1.4 Literature study

Since the problem is relatively new and unique for the web, my main literature will be web articles and W3C mailing lists. There are many experienced individuals writing good articles about the problem and mostly discusses how an API can look like. The W3C mailing lists are a good resource to get insight into how the browser vendors view the problem. To investigate third-party implementation approaches, I will read the implementation attempts that I can find. To understand the issues with implementing the

problem better, I will read books, scientific papers and articles covering how rendering engines work. To solve the circular problems of the third-part implementation I will read scientific papers and articles how other languages have solved similar problems. To make sure I fully understand responsive web development I will read books about the subject. To support my claims that modular development is a good thing, I will research scientific papers and books.

1.4.1 Current literature

The following is the literature that has been gathered so far.

Books

- Tim Kadlec. Implementing Responsive Design: Building sites for an anywhere, everywhere web. New Riders, 2012.
- Frain, Ben. Responsive Web Design with HTML5 and CSS3. Packt Publishing, 2012.
- Scott Jehl. Responsible Responsive Design. A Book Apart, 2014.

Scientific papers and reports

- Jerjas, Allan. Building Blocks of Responsive Web Design. KTH, 2013. URN: `urn:nbn:se:kth:diva-142345`
- Tobias Sundqvist, Christoffer Wåhlander. Applikationsutveckling i Responsive Web Design. LIU, 2013. ISRN: `LIU-IDA/LITH-EX-G--13/011--SE`
- Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, Rastislav Bodík. Parallelizing the Web Browser. In Proceedings of the First USENIX Workshop on Hot Topics in Parallelism. Department of Computer Science, University of California, Berkeley, 2009.
- Calin Cascaval, Seth Fowler, Pablo Montesinos Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, Vrajesh Bhavsar. ZOOMM: A Parallel Web Browser Engine for Multicore Mobile Devices. In Principles and Practice of Parallel Programming. Qualcomm Research Silicon Valley, 2013.
- Alan Grosskurth, Michael W. Godfrey. Architecture and evolution of the modern web browser. In Proceedings of the 21st IEEE international conference on software maintenance (ICSM'05). David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada, 2006.
- Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, Herman Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In USENIX security symposium (Vol. 28). Microsoft Research, University of Illinois at Urbana-Champaign, University of Washington, 2009.

- Lingjun Fan, Weisong Shi, Shibin Tang, Chenggang Yan2 Dongrui Fan. Optimizing web browser on many-core architectures. In Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference. Chinese Academy of Sciences, Wayne State University, 2011.
- Carmen Badea, Mohammad R. Haghighat, Alexandru Nicolau, Alexander V. Veidenbaum. Towards Parallelizing the Layout Engine of Firefox. In Proceedings of the 2nd USENIX conference on Hot topics in parallelism. UC Irvine, Intel Corporation, 2010.
- Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, Shriram Krishnamurthi. Flapjax: a programming language for Ajax applications. In ACM SIGPLAN Notices (Vol. 44, No. 10, pp. 1-20). 2009, October.
- G. J. Badros, A. Borning, K. Marriott, P. Stuckey. Constraint cascading style sheets for the web. In Proceedings of the 12th annual ACM symposium on User interface software and technology (pp. 73-82). ACM. 1999, November.

Articels

- Tali Garsiel, Paul Irish. How Browsers Work: Behind the scenes of modern web browsers. html5rocks, 2011. `http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/`
- Dave Hyatt. Layout and Rendering. The WebKit Open Source Project, 2007. `http://www.webkit.org/projects/layout/index.html`

Other

- Francois Remy. The :min-width/:max-width pseudo-classes. W3C mailing list, 2013. `https://lists.w3.org/Archives/Public/www-style/2013Mar/0368.html`
- Kevin Mack. Element Query and Selector Media Query Types. W3C mailing list, 2014. `https://lists.w3.org/Archives/Public/public-respimg/2014Oct/0005.html`
- Responsive Issues Community Group element queries use discussion repository. `https://github.com/ResponsiveImagesCG/eq-usecases`. Creating the use cases draft: `https://responsiveimagescg.github.io/eq-usecases/`

In addition to the listed literature, numerous web articles has been found regarding element queries. There are around 10 unique open source implementation attempts of element queries to examine. No good in-depth literature regarding rendering engines and why element queries are hard to implement natively has yet to be found.

The literature study will be examined by a written report.

1.5 Related work

The CCSS language proposed in the *Constraint cascading style sheets for the web* report provides a formal specification to CSS (and extensions) which element queries would be easy to build upon. It is not directly stated in the report that element queries are supported (since only a subset of CSS has been described), but the language construction itself is appealing to element queries. One of the objectives to CCSS is to enable CSS selectors to arbitrary elements, not only parent elements. That, along with the powerful constraints syntax makes it a good environment for element queries. However, one of the premises for the thesis is to make element queries work in existing browsers. Since CCSS is not a recommended standard today (or even fully constructed), the CCSS ground will only be interesting to build upon on a theoretical level. If element queries could be expressed in the syntax of CCSS, it could be easier to understand the problems and implications of element queries since it then exists a formal description of it. Since the CSS language itself doesn't impose any cyclic rules, it is unclear how one would handle cyclic element queries in CCSS. Due to the behavior of the constraints solver algorithm (merging the selector matching and layouting phase) a cycle can easily be identified when the solver doesn't find any solutions.

The Flapjax language descibed in *Flapjax: a programming language for Ajax applications* is relevant due to it's handling of cyclic data flows. They describe various algorithms to detect cycles in directed graphs. Since element queries will indirectly create directed style dependency graphs, the algorithms described in the paper can be useful. Cycle detection in element queries may differ in the sense that the whole graph is not controllable (as the graph exists both in JavaScript and CSS) which means that the whole graph cannot be traversed. One way to overcome this limitation would be to parse the CSS and construct the whole rule dependency graph in JavaScript, thus making it fully traversable. This way the algorithms described in the Flapjax paper can be applied. However, it might be a better option to simply detect cycles by letting the cycles loop an amount of times and have the element query framework detect that the same rules has been applied back and forth frequently.

There are numerous informal and unofficial implementation attempts and discussions about element queries, which of course will be of great use. The RICG will be a good resource to read and discuss how native element queries would look like. The mailing lists of W3C will also be a good resource to gain insight into the performance problems that native element queries might bring, as well as a deeper understanding of rendering engines.

1.6 Methodology

A theoretical study of the problem and rendering engines will be performed in order to understand the premises and limitations of implementing element queries in browsers. When a deep understanding of the problem has been acquired, a third-party framework shall be developed. A study of implementation attempts will be made and a comparison

of approaches will be presented. Then the API will be designed and implemented. Studies of the subproblems that may arise during the implementation will be studied in parallel to the implementation.

1.7 Boundaries

The focus of the thesis lays on developing a third-party framework that realizes element queries. All theoretical studies and work will be performed to support the development of the framework.

1.7.1 What will be done

- A third-party implementation of element queries will be developed.
- The problems of implementing element queries natively will be addressed.
- Theory about rendering engines, CSS, HTML and responsive web design will be given to fully understand the problem.

1.7.2 What will not be done

- No efforts will be made to solve the problems accompanied with a native solution.
- No API or similar will be designed for a native solution.
- UI and UX design will not be addressed, other than necessary for understanding the problem.
- No complete history of browsers, the Internet or responsive web design will be given other than neccesary.

1.8 Timetable

The work can be divided into 4 stages, each representing a milestone. Report writing will occur in parallel to all the stages, as shown in figure A.1 in the appendix. The stages are:

1. **Literature study:** Search for suitable literature and sources will be performed in 2 weeks (a big part of the literature study has already been done before the writing of this document). Total time of literature study is estimated to 4 weeks.
2. **Background:** The background of the problem will be investigated by reading the literature. The background will mainly address responsive web design and modular development.

3. **Theory:** In this part the theory will be addressed. Here I will find out what element queries really are and how they relate to CSS and HTML. The most time will be spent investigating how rendering engines work and why element queries are hard to implement.

4. **Third-party framework:** Here an analyze will be done of different implementation approaches and other implementation attempts will be investigated. The larger part of this stage will be dedicated to designing an API and implementing the actual framework.

Appendix A

Resources

A.1 Timetable

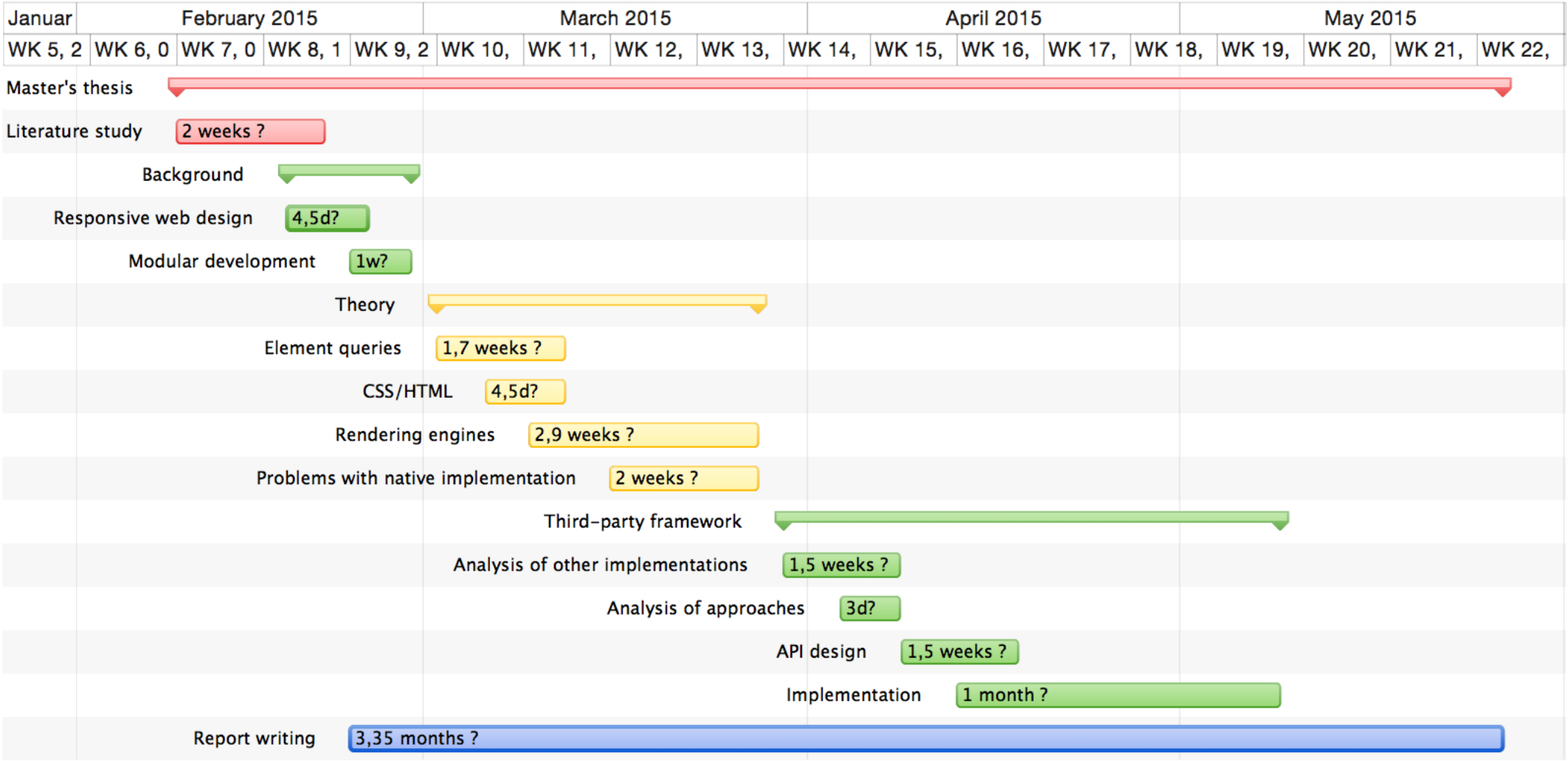


Figure A.1. Box plot of number of positions sent per iteration using this scheme

A.2 Practical problem formulation document

Problem formulation

A very simple example that will describe the essence of the problem.

What we want

We want to develop a responsive module that will respond and react to dimension changes of *the module*. The module that we want to develop will be named*simple-module* and should be composable in any layout configurations.

simple-module

The HTML code for the module is the following:

```
<div class="very-simple">
    <h1>Simple module</h1>
    <h4 class="show-big">
        This text will only be shown when module is <b>big</b>.
    </h4>
    <h4 class="show-small">
        This text will only be shown when module is <b>small</b>.
    </h4>
    <p>Will be colored red when small and blue when big.</p>
</div>
```

The requirements for the module are:

1. When the module has <= 500px of width, the `show-big` class should hide the target elements.
2. When the module has > 500px of width, the `show-small` class should hide the target elements.
3. When the module has <= 500px of width, the background of the module should be colored red.
4. When the module has > 500px of width, the background of the module should be colored blue.
5. **The module should be responsible for fulfilling the above requirements by itself.**

What is meant with `When the module has ... of width` is that when the outer div `very-simple` is allowed by the container element to expand to the stated amount of width.

What we need in order to fulfill the requirements of the module,*is to have conditional CSS*

rules to style elements differently by the container dimensions.

Approach 1: media queries

By using media queries, we can apply conditional CSS rules by the viewport dimension like so:

```
@media (max-width: 500px) {
    .very-simple {
        background: red;
    }

    .show-big {
        display: none;
    }

    .show-small {
        display: block;
    }
}

@media (min-width: 501px) {
    .very-simple {
        background: blue;
    }

    .show-big {
        display: block;
    }

    .show-small {
        display: none;
    }
}
```

Test

Let's test the module that we have created with approach 1. Here we see a test application that uses 4 "instances" of the simple-module module. One is placed in the top and allowed to expand freely in width. The other 3 are given 31% of the viewport width, and are centered horizontally.

Following is the interesting code for the example application. Don't mind the special syntax. The important thing is that the simple-module CSS and HTML is loaded from the module, not defined in the app.

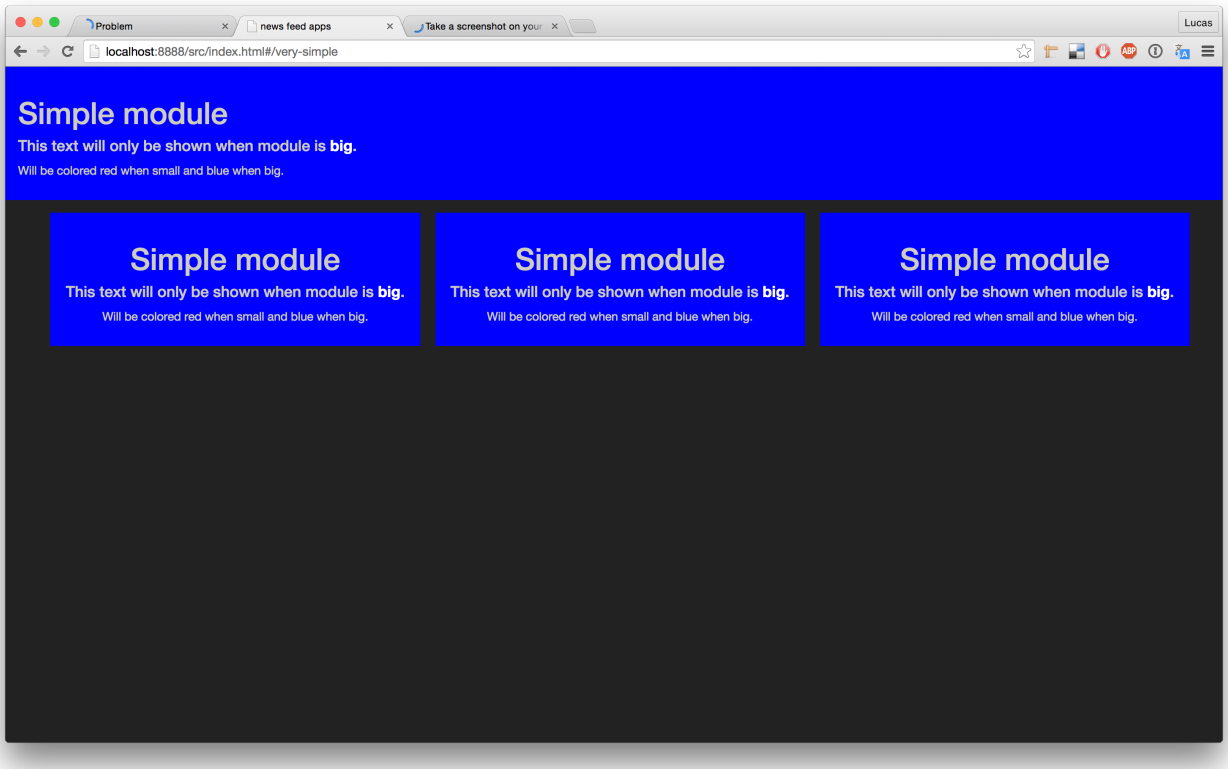
```
<div>
    <very-simple></very-simple>
</div>
<div class="row">
    <div class="col"><very-simple></very-simple></div>
    <div class="col"><very-simple></very-simple></div>
    <div class="col"><very-simple></very-simple></div>
</div>
```

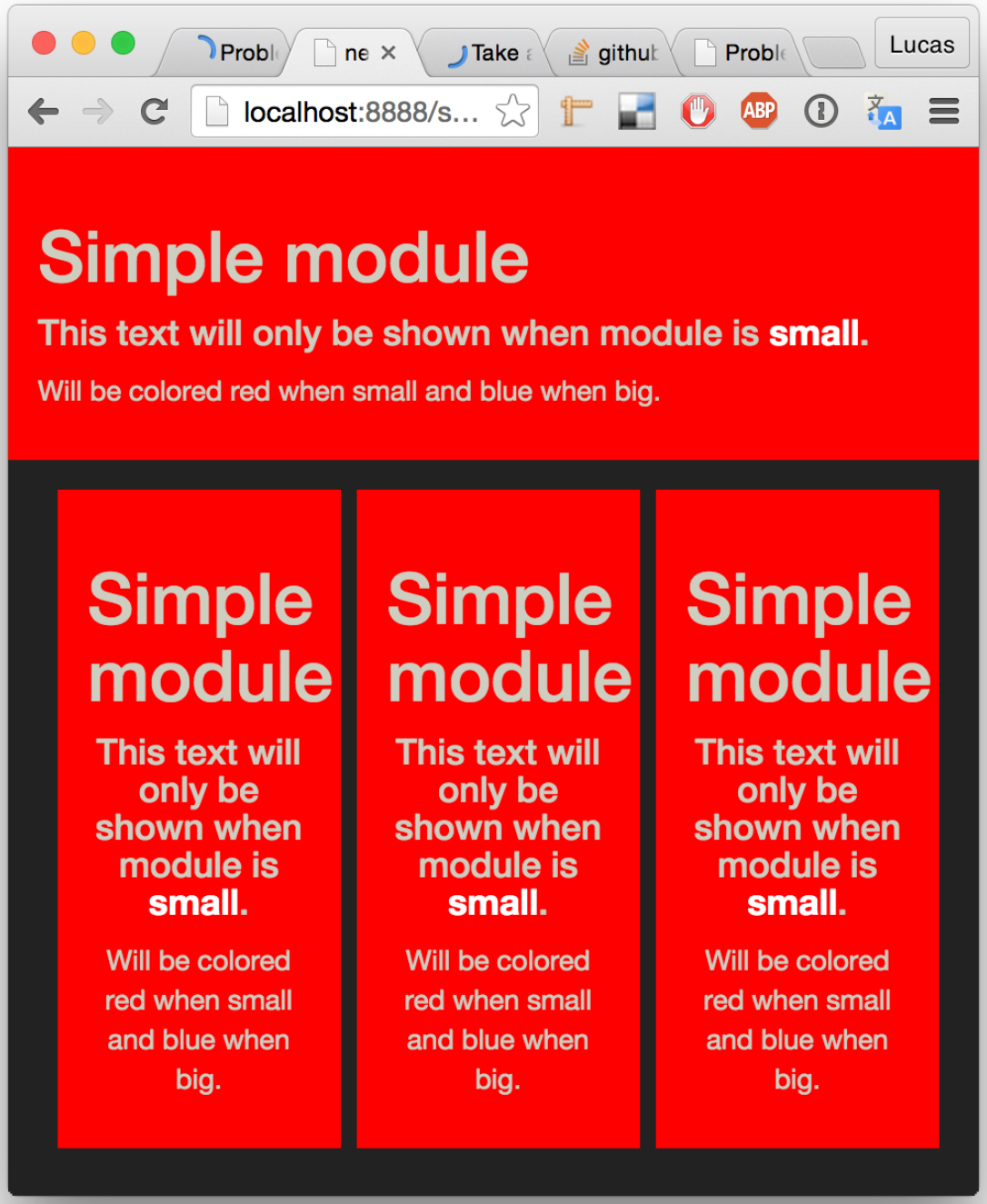
```
@import "../../modules/very-simple/very-simple.css";

.row {
    margin: 15px;
    text-align: center;
}

.row .col {
    width: 31%;
    margin-left: 1%;
    display: inline-block;
}
```

This results in the following:





The module placed at the top satisfies all requirements, since it reacts to the style breakpoint at 500px. However, the three instances beneth it are all broken. In the first picture the individual width of the lower modules are 440px, so the modules should be colored red and display the other text. As we can see in picture 2, the lower modules only change style state when the upper also changes state. *This is because all modules style themself relative to the viewport, disregarding the actual size that they have.*

So this approach works properly when the module is allowed to expand its width to the whole viewport. If the dimension of the module is constrained by the user somehow, this approach will break.

We failed to create the described module with this approach, since the module is not layout-agnostic. In other words, the module is not composible in any layout.

Approach 2: element queries framework

By using the third-party *element queries* framework, we can satisfy all requirements of the simple-module. The framework (named ELQ and will appear as `elq` in the code) will be plugin-based, so there will probably be different syntaxes for different app requirements. Let's consider this simple example, which will use the *breakpoints* plugin (`elq-breakpoints` in code).

The `elq-breakpoints` plugin is able to listen to arbitrary elements for dimension changes, and will apply classes to the elements to reflect the dimension changes. Consider the following example:

```
<div elq elq-breakpoints elq-breakpoints-width="300 500 800">
    <h1>I can do element queries!</h1>
</div>
```

The div that we want to perform element queries on has received three special attributes. The first attribute `elq` tells the framework that the element and all the children of it will be targeted for element queries. The second attribute `elq-breakpoints` tells the framework that this element should use the breakpoints plugin. The breakpoints plugin will read the third attribute `elq-breakpoints-width` and will then know that the module want to use the width breakpoints of 300, 500 and 800 pixels for element queries.

Then, what will happen is that the framework will listen to the element for dimension changes, and then add appropriate classes to the element. If the element has the width of 550 pixels, it will have the following classes:

- `elq-width-above-300`
- `elq-width-above-500`
- `elq-width-under-800`

Of course, the height works in the same way. Now let's see how this can be used to make the simple-module behave as we want.

The HTML of the module will look like this:

```
<div class="very-simple" elq elq-breakpoints elq-breakpoints-width="500">
    <h1>Simple module</h1>
    <h4 class="show-big">
        This text will only be shown when module is <b>big</b>.
    </h4>
    <h4 class="show-small">
        This text will only be shown when module is <b>small</b>.
    </h4>
    <p>Will be colored red when small and blue when big.</p>
</div>
```

And the CSS of the module will look like this:

```
.very-simple.elq-width-under-500 {
    background: red;
}

.very-simple.elq-width-under-500 .show-big {
    display: none;
}

.very-simple.elq-width-under-500 .show-small {
    display: block;
}

.very-simple.elq-width-above-500 {
    background: blue;
}

.very-simple.elq-width-above-500 .show-big {
    display: block;

.very-simple.elq-width-above-500 .show-small {
    display: none;
}
```

Of course by using a CSS preprocessor such as LESS or SASS, the CSS can instead be written like this:

```
.very-simple.elq-width-under-500 {
    background: red;

    .show-big {
        display: none;
    }

    .show-small {
        display: block;
    }
}

.very-simple.elq-width-above-500 {
    background: blue;

    .show-big {
        display: block;
    }

    .show-small {
        display: none;
    }
}
```

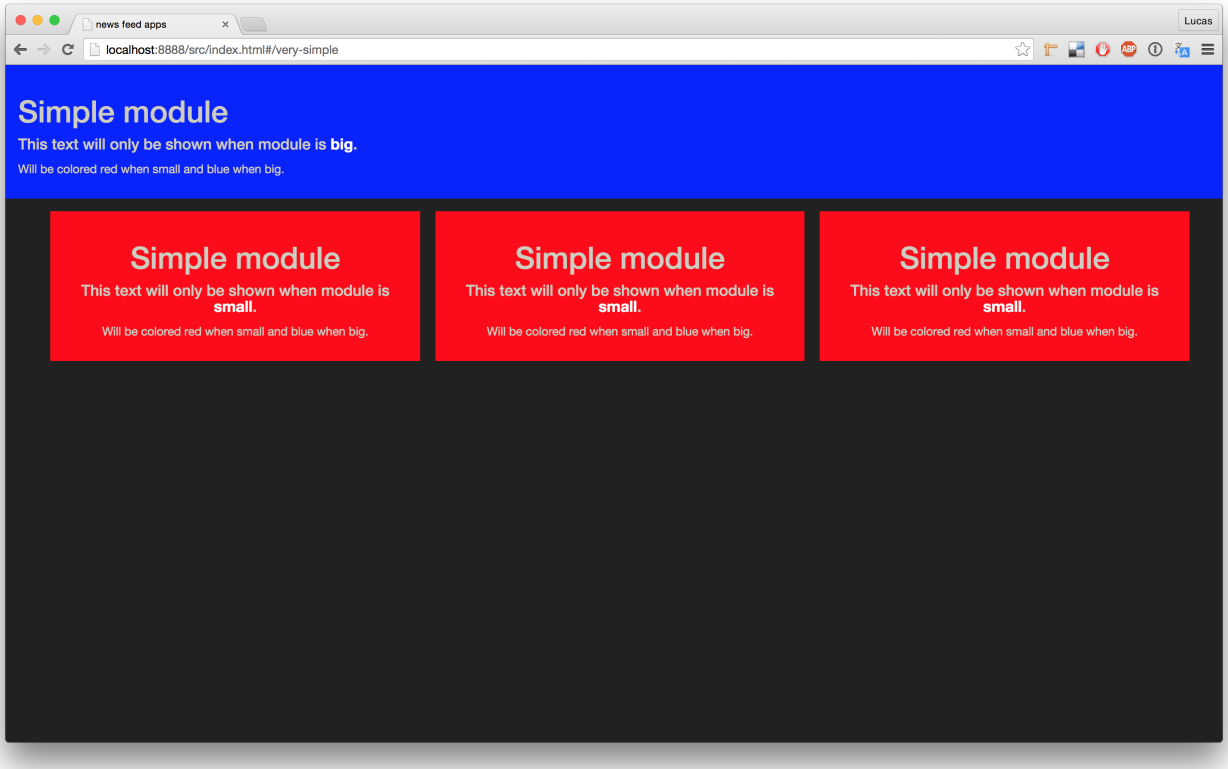
Which is very similar to the media query approach. Only here, we target the `.very-simple` element for queries, instead of the viewport.

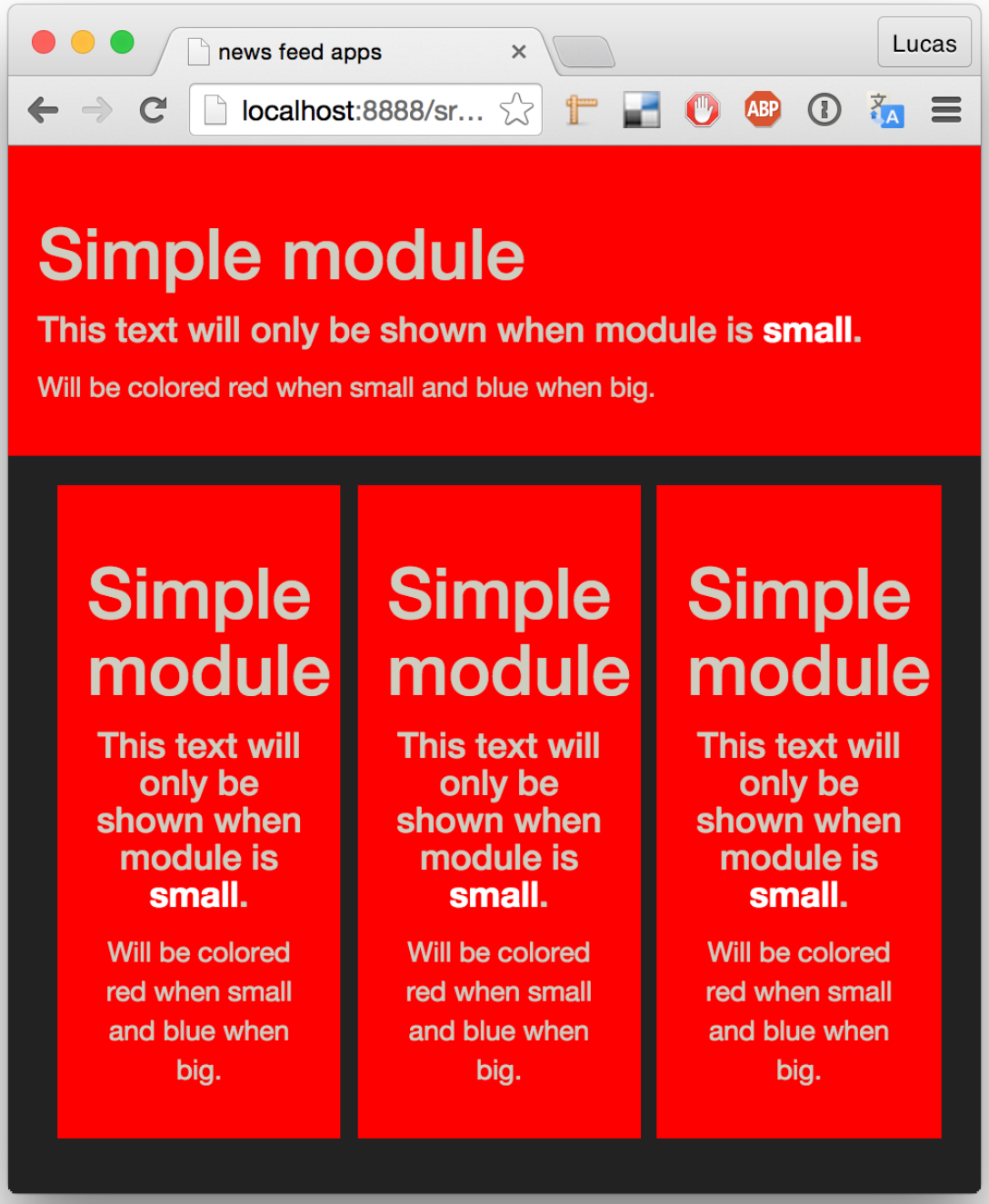
Test

Let's test the module with the new approach. The app layout will be the same, but we of course need to include the framework in the app in order for the modules to work. Apart from including the framework, the developer of the app do not need to change anything. Including the framework can be done like so:

```
<script src="elq.js"></script>
<script src="elq-breakpoints.js"></script>
```

Our app then looks like this:


news feed apps
Lucas
localhost:8888/src/index.html#/very-simple
Simple module
This text will only be shown when module is big.
Will be colored red when small and blue when big.
Simple module
This text will only be shown when module is small.
Will be colored red when small and blue when big.
Simple module
This text will only be shown when module is small.
Will be colored red when small and blue when big.
Simple module
This text will only be shown when module is small.
Will be colored red when small and blue when big.



As we can see, the module behaves exactly as we want it to. All the requirements are satisfied.

Conclusion

As illustrated in this document, it is impossible to write responsive modules that are encapsulated and layout-agnostic with media queries. Since media queries is the only native way of having responsive elements, there is clearly a need for native element queries. Because native element queries impose complications to browser implementations and

complexity to the CSS language, it will probably be a very long time before we will see native element queries (if it will ever happen). By crafting a third-party framework for element queries (a prolyfill) we can create such modules today.

All the images in this document that shows the simple-module module are screenshots of an actual working example powered by the experimental build of the ELQ framework.

A.3 Practical cyclic rules discussion document

Cyclic Rules

This is a very simple document that will describe the cyclic rules that may occur with element queries. The examples will use the syntax of the ELQ framework as described in the Problem Formulation document. Also, the CSS presented in this document is written in preprocessor syntax to make the examples more clear, also as described in the problem formulation document.

Direct cyclic rules

The most straight forward occurances of cyclic rules are when a programmer has specified conflicting width constraints and width updates for an element.

Example 1

```
<div id="container" elq elq-breakpoints elq-breakpoints-width="300 500">
    <div id="child"></div>
</div>
```

```
#container {
    width: 250px;
}

#container.elq-width-under-300 {
    width: 550px;
}

#container.elq-width-above-500 {
    width: 250px;
}
```

The following will happen for the module:

1. If the initial width of the container is between 300px and 500px, the selector `#container` will be the only matching selector and the width of the container element will be set to 250px. Then next step will be 2. If the container element is under 300px wide, next step will also be 2. If the container element is above 500px wide, next step will be 3.
2. The width of the element is under 300px, and the selector `#container.elq-width-under-300` will now match and since it is more specific than the `#container` selector, the new width of the element will be 550px. Next step will always be 3.
3. The width of the element is above 500px, and the selector `#container.elq-width-`

`above-500` will now match and since it is more specific than the `#container` selector, the new width of the element will be 250px. Next step will always be 2.

Clearly, the browser will be stuck in an infinite layout cycle pending back and forth between case 2 and case 3 (250px and 550px). Depending on CSS implementation, this may or may not cause a rendering engine crash. What could happen is that the rendering engine will execute one layout pass and then evaluate the next set of matched selectors and so on, which will lead to a functioning site but since a relayout is enforced every update, the performance impact will be huge.

Indirect cyclic rules

Cyclic rules can also occur in a more indirect way. For instance, if the container element will match the width of the child, but the child changes width depending on the parent width, a cycle might occur.

Example 2

```
<div id="container" elq elq-breakpoints elq-breakpoints-width="300 500">
    <div id="child"></div>
</div>
```

```
#container {
    display: inline-block; //<- Will match the width of the child element.
}

#child {
    width: 250px;
}

#container.elq-width-under-300 #child {
    width: 550px;
}

#container.elq-width-above-500 #child {
    width: 250px;
}
```

What we have here is the same situation as in example 1. What makes this situation trickier is that it is less obvious for numerous reasons:

- The rules of the child element and the rules of the container element might be separated into different parts of the stylesheet (or even different stylesheets). The problem cannot be found without considering both of the rulesets.
- The child element might be another module, and by adding one line to the parent

module (the container) a cycle has appeared.
- The programmer has to be well aware of how the `display` property affects the element and what implications the `inline-block` has to the element.

The examples given so far have been very simple and easily detectable. However, cycles can occur in a much more complex way. As of writing this, it is still unclear to me what would be a good example for a very complex cycle occurance. However, my personal theory is that as factors that creates the cycles increases the harder it gets to detect them (as a programmer). To clarify, let's consider the factors of example 1 and 2:

- **Example 1:** This cycle is very easy to spot (as reading the CSS out loud almost gives away the problem in English). This example only operates on the width of the container element, which is 1 factor to the cycle.
- **Example 2:** This cycle is easy to spot, but not as easy as example 1. The reason is that we cannot know that there is a cycle unless we also consider the container element properties (here the `` `display` `` property). So this cycle depends on the width of the child element, the width of the container element and also the display property of the container element. So this example has 3 factors to the cycle.

Let's consider another example which has even more factors:

Example 3

```
<div id="container" elq elq-breakpoints elq-breakpoints-width="300 500">
    <div id="child">
        When in doubt, mumble.
    </div>
</div>
```

```
#container {
    display: inline-block; //<- Will match the width of the child element.
}

#child {
    font-size: 1.5em;
}

#container.elq-width-under-300 #child {
    font-size: 2em;
}

#container.elq-width-above-500 #child {
    font-size: 1em;
}
```

In this example, it's virtually impossible to tell if the rules are cyclic. So what happens here is that the container element will get the size of the child element, which width depend on the text inside it. The width of the text depends on the font size, which is initially set to 1.5em. The `em` unit is relative to the inherited font size of the element. So basically the width of the container element is dependent on the inherited font size. When the container element is below 300px, the font size is increased to 2em and if the element is above 500px the font size is reduced to 1em. So if 2em results in a font size big enough to make the child element bigger than 500 pixels, and if 1em results in a font size small enough to make the child element smaller than 300 pixels we have a cycle.

Let's list the factors that creates the cycle:

- The width of the container element
- The display type of the container element.
- The font size unit of the child element.
- **The inherited font size of the child element.**
- **The text of the element.**

This adds up to 5 factors. However, the bolded factors are of a far worse type than the ones we have discussed until now. They are impossible to determine at parse time. In this example the text is static but it could be added dynamically. Also, the inherited font size of the child depends on the closest ancestor with a font size property defined. However, since ancestors also can have their font sizes defined in relative units, the dependency tree can go to the root of the document. Further, if no ancestor defines an absolute value for the font size it is up to the browser to default to a size, which is impossible to reason about at parse time. This means that there is no way of telling if the cycle will appear or not without actually running the code. Also, the cycle may appear in different browsers and settings, which makes the cycle even harder to detect.

Far worse is that the previous examples have been of the character of which the reader thinks "why would anyone ever want to do that?", which gives the false hope that even though it is possible to create cycles - developers would never encouter it since "real" code would not be close to the stupid examples given here. With example 3, this is no longer true. Increasing the font size when the layout area is smaller and decreasing the font size when the layout area is bigger is something that is frequently done. Think about how websites adapts to small and big screens. Hint: the font size is usually bigger on the mobile phone than it is on the desktop.

Conclusion

In short, the programmer needs to be more alert when developing with element queries in order to avoid cycles. One could argue that the programmer still would need to be alert and cunning while styling sites to avoid errors even without element queries. While this is

technically true, the difference is that the damange that programmers may cause due to styling errors are very different. With element queries, it is possible to create infinite cycles that cripples the rendering engine. Without element queries, the worst that could happen due to styling errors is that the website simply doesn't look good. So element queries really put more responsibility to the developer and demands a deeper understanding of CSS. In order to aid and guide the programmer, it is important to be able to catch and handle cyclic rules when implementing element queries.

It is possible to detect the simplest forms of cycles but a robust cycle detection mechanism must be executed at runtime in order to catch the more complicated cycles. Another advantage of detecting the cycles at runtime is that the detection is CSS and implementation agnostic, since the detection system does not need to know all the mechanics of styling - it just needs to know if the browser is stuck in a cycle or not.