Department of Computer Science

Submitted in part fulfilment for the degree of MEng

A Domain-specific Language for HTML and CSS Linting

Marcin Piskorz

30 April 2019

Supervisor: Dimitris Kolovos

STATEMENT OF ETHICS

This project will be carried out in accordance with the ethical and academic principles that have been set. All works from other parties will be cited and listed in the bibliography as appropriate. All work seen in the report and in my project is my own, unless stated.

TABLE OF CONTENTS

TABLE OF FIGURES

Executive summary

Frontend frameworks offering complex CSS style sheets to web developers are getting more and more popular and their number is still growing. Every framework introduces their own set of rules that must be obeyed to produce correct documents on top of the requirements of the standard HTML structure.

Unfortunately there are not many tools which are capable of automating the validation process of a HTML document utilising a certain framework. Without such tools the creators are forced to ensure correct use of the framework features on their own, taking a portion of their time which they could spend implementing the actual design of the website.

The motivation of this project is to have an impact on this issue by introducing a simpler way to produce linting tools for the family of the CSS frameworks. At the moment the only way to develop such a tool is to write a linter from scratch which only applies to the rules of one particular framework.

If we analyze the frameworks we can see that all of them introduce only a limited set of rules with different logic behind them, which are reused multiple times with varying parameters to represent the complete set of requirements for a specific framework. Due to this fact the validation rules can be restricted to a limited domain, which leads us to the title of this project.

The main aim of this project is to design and implement a domain specific language capable of describing every validation rule that may be introduced to HTML documents by a modern CSS framework. The secondary aim is to create a feature which can generate an executable program capable of validating a given HTML document based on the rules described in the developed language.

The design of the language aims to include concepts that can describe a set of types for validation rules and allow the user to personalize them with editable parameters. The proposed syntax for the language includes explanation of the actual rule used in plain English.

To implement the design I have used already existing software called Meta Programming System (MPS) from JetBrains, which allows for creating own domain specific languages. On top of the essential features for describing abstract and concrete syntaxes of

languages, it also provides ways to create a text editor with features that are related to the developed language. Another useful element of MPS that I have used was the model-to-text generator which allowed me to translate rules described in my own language to a general purpose language that can be executed to perform the validation. Thanks to using a well supported technology for developing domain specific languages I was able to focus on the quality of the actual language.

The generator translates the linter definition from my language to JavaScript which I have chosen due to the fact that it can be run directly from the browser, meaning it is capable of validating dynamic websites at various states. Once the code is generated it can be extended by the user if needed.

My solution is available to anyone through a repository which is available online[19]. I provide documentation of my language as well as a complete guide explaining setup and features of each components involved in the implementation.

To evaluate my solution I have attempted to represent validation rules of an existing linter tool for one of the frameworks with use of my language. Only 2 out of 52 rules could not be described with use of the concepts of my domain specific language meaning that it covers over 96% of the domain. I have then created a test page including 50 violations on both the original liner as well as the generated one which aims to clone the behaviour of the authentic version. The generated linter managed to pick up the same errors as the original tool. During the translation from an existing solution I have managed to find minor issues with the language.

The project can be considered an overall success as it proves that model driven engineering is a viable way to make creation of linters easier. With some further work and additional evaluation my solution could persuade the developers of CSS frameworks to implement linters for their creations.

The project does not involve any legal, social or professional issues. Any software used to implement my solutions does not require paid licences.

1 Introduction

1.1 Background

Ever since the beginning of the World Wide Web, Hypertext Markup Language (HTML) has been the standard language for developing web pages. Initially its only task was to describe the structure of the document which is then rendered by a browser As the years have passed more requirements in the design area have appeared. To meet them, the language was upgraded with a more sophisticated supply of attributes allowing to define colours, topography, add media content and more. This led to increase of the complexity of HTML documents, as the result reducing their readability and maintainability.

To bring the discussion back to web development Cascading Style Sheets (CSS) have been introduced together with the ways to link it with HTML elements, allowing the programmers to separate the styles from the structure of the website. Due to the fact that both of these languages are descriptive, it is crucial to have an effective way to validate correctness of the code, as the browser will not notify us about the syntax errors and ignore any content that it was unable to parse or result in rendering the content in an unintended fashion.

This is usually done by a linting tool plugin implemented in a development environment, which greatly facilitates the developer's work by providing the integrated development environment (IDE) with the information needed for highlighting the faults, proposing potential ways to fix them in real time and furthermore enhancing the document structure by auto-applying different spacing, colours and font weights to substrings representing relevant parts.

Even though HTML and CSS are considered to be completely separate, more recent front-end frameworks such as Bootstrap, Skeleton or Foundation tend to link the concept of the structure of HTML code with the classes used by the elements, therefore introducing new potential ways to produce invalid arrangements of which the traditional linters are unaware. In order to back up the usability of the frameworks, some framework-specific linting tools have been developed, such as Bootlint written in JavaScript for validating websites using Bootstrap. The industry is still lacking general purpose applications which would allow for implementation of validation rules for similar concepts used across all frameworks using different keywords and only adding additional rules on top of them. This would allow for quicker development of specific linters for the frameworks which are missing linting tools and the ones to be yet produced.

1.2 Aims

The main aim of this project is to create a domain-specific language for describing implicit styling rules of front-end frameworks which introduce additional structural requirements for the correctness of HTML documents. In order to backup the capability and quality of the language itself, I will produce a validation model, containing all of the rules introduced by one of the frameworks, together with a linting tool which will take it as one of the inputs.

The final application should be proficient in validating given HTML and CSS code implementing the a specific framework, as defined by the input and outputting the contained errors, therefore proving that it can be used to produce linting tools for other frameworks of the same sort. For this project I am going to assume that any HTML or CSS code fed to the application for quality measurements is structurally correct in regards to HTML.

1.3 Report structure

The report will be divided into seven chapters discussing the preparation and the actions that I have taken in order to reach the aims of the project. The purpose of this chapter is to give a brief introduction and background on the subjects related to the project and present the main goals to be reached. The next chapter i.e. the literature review will be a summary of the relevant literature research, which is the prerequisite knowledge needed for the reader in order to fully understand the problem of this project and the past solution approaches. Chapter number 3 will be a problem analysis, focusing on the aims in more detail as well as presenting the tools and the issues that will need to be overcome. It will also discuss the potential solutions and tools that may be used in the development. Next, the design chapter will follow, explaining all of the decisions regarding the architecture and style, as well as their justification. The fifth chapter will target the actual implementation of the solution for the project. I will explain and justify each step that I make in detail, underlining the most important principles of the solution. Chapter 6 will be a complete evaluation of my solution for this project, checking if it has met all of the requirements listed in section 3 of the report as well as compared against already existing, although less general solutions. Finally the concludinng chapter will act as a summary for the entire project, including the suggestions for further work that could be done in the field of linting solutions.

2 Literature review

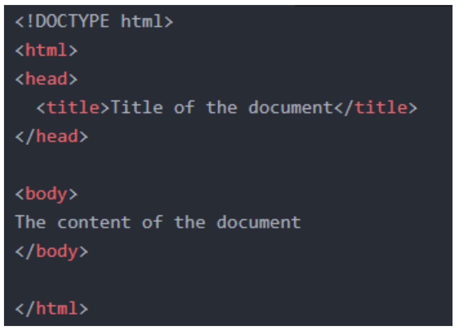
The following chapter of my report will review the literature concerning the fields relevant to my project. There are several topics that the reader must understand in order to interpret the project's problem correctly. First of all, it is required to have the fundamental knowledge about HTML and CSS languages and their structure as well as the ability to recognize errors in the documents of each type. I will cover this by gathering information from various sources regarding these languages. I will also provide several code examples with description to explain the common errors made by web developers. Next I am going to describe what is the task of linters and their importance in the programming industry as well as to analyze how they work behind the scenes. The knowledge about domain-specific languages and their use cases will also be covered in this section. Finally, I will investigate some front-end frameworks under the angle of the additional validation rules that they introduce and examine existing linting solutions for them. To conclude my literature review, I will summarize my research and discuss the discovered ideas that I could implement in my own solution.

2.1 Hypertext Markup Language (HTML)

HTML is the World Wide Web's core markup language. Initially it was designed for semantical description of scientific documents. The primary design was then adopted to describe other types of documents and even applications[1]. Now a HTML document is supposed to provide the browser with all the necessary data to render a web page.

2.1.1 Document structure and syntax

HTML documents consist of a tree of elements and text. An element is denoted by a start tag such as "<div>", some text content or another element, and finally an according end tag, such as "</div>" (for some elements only the start tag is sufficient as they cannot have any inner content, for example the line break element denoted by "<br>"). Tags have to be nested such that elements are all completely within each other, without overlapping.



```
<!DOCTYPE html>
<html>
<head>
  <title>Title of the document</title>
</head>

<body>
The content of the document
</body>

</html>
```

Figure 2.1 Basic HTML document structure.

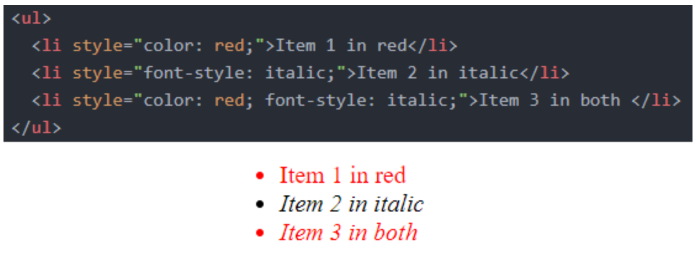
In order for the browser to correctly identify an HTML document, it must contain a document type declaration denoted by “<!DOCTYPE html>”, which is placed at the very top of the document. Every HTML document must also contain the main html tag including two child tags: a head tag and a body tag, the head element being a container for all metadata elements feeding the browser with useful information about the page, and a body tag containing all of the actual content to be displayed. A basic example structure of a semantically correct HTML document can be seen in figure 2.1.

2.1.2 HTML elements

As mentioned before, elements are the building blocks of the documents. Every type of element has a different purpose, and some of them can only appear as a child of a concrete set of element types. For example the list item denoted by “<li>” tag, can only be a child of an ordered list(“<ol>”), unordered list(“<ul>”), and a menu list (“<menu>”).

Each element shares a set of properties common to the element class, and holds another set specific for that element, controlling its behavior. Altering the default values of those properties can be done by using scripts or by redefining them as attributes inside the starting tag of an element. A standard attribute declaration consists of the attributes name, followed by an equal sign and a desired value inside quotes. Each element has a set of attributes with default values that can be altered by redefining it in the opening tag.

One of the most important attributes regarding this project is the style attribute, allowing to the styling of a particular element to be specified. It has many properties on its own, hence it is possible to pass a list of style definitions. The syntax of a style definition is as follows: “style_property:value;”, where the semicolon separates it from another definition.



```
<ul>
  <li style="color: red;">Item 1 in red</li>
  <li style="font-style: italic;">Item 2 in italic</li>
  <li style="color: red; font-style: italic;">Item 3 in both </li>
</ul>
```

- Item 1 in red
- *Item 2 in italic*
- *Item 3 in both*

Figure 2.2 HTML elements example.

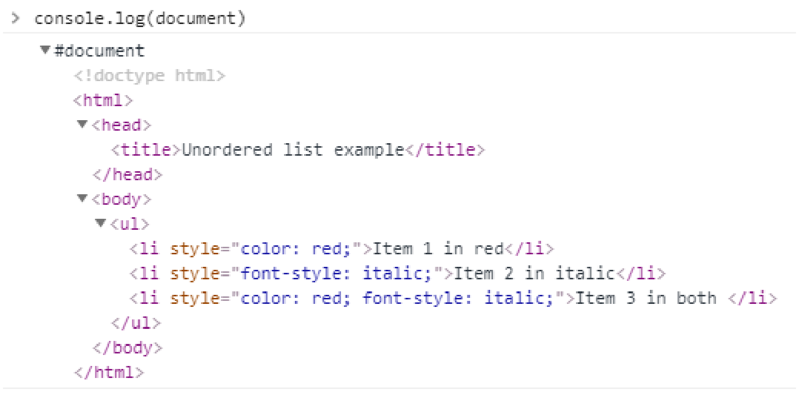
To sum up the element declaration, the figure 2.2 shows an example code representation of an unordered list containing three elements with different styling followed by an actual web page content to be rendered by a browser.

2.1.3 Document Object Model (DOM)

Once a HTML document loads into a scriptable browser, the browser parses it into a hidden, internal roadmap of all the elements it recognizes as a scriptable object[2]. This roadmap is hierarchical in nature creating an object oriented tree structure with the document (“<html>”), the most “global” object as the root.

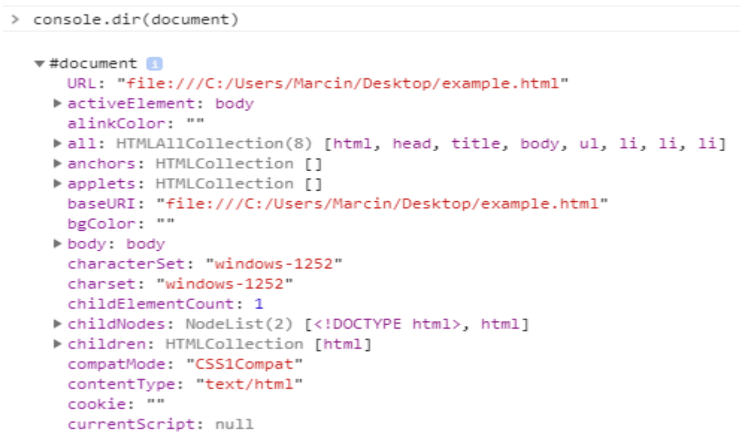
Each of the elements of a DOM tree is represented by an object with its own API, so that they can be manipulated by scripts written in JavaScript, allowing to change the content of the page dynamically, for instance depending on the users’ input or another application interacting with the page. Scripts can be embedded on a page using the “<script>” element and putting the script directly between the end tag or passing a reference to another file containing the scripts via a “href” attribute.

In the figure 2.3 we can see an example of a DOM tree printed by the browsers’ console. Figure 2.4 shows the document (“<html>”) element itself interpreted as an object including only some of its properties.



```
> console.log(document)
  ▼#document
      <!doctype html>
      <html>
      ▼<head>
          <title>Unordered list example</title>
        </head>
      ▼<body>
        ▼<ul>
            <li style="color: red;">Item 1 in red</li>
            <li style="font-style: italic;">Item 2 in italic</li>
            <li style="color: red; font-style: italic;">Item 3 in both </li>
          </ul>
        </body>
      </html>
```

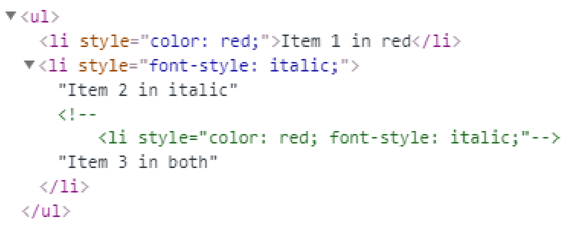
Figure 2.3 Document DOM tree.



```
> console.dir(document)

  ▼#document
      URL: "file:///C:/Users/Marcin/Desktop/example.html"
    ▶ activeElement: body
      alinkColor: ""
    ▶ all: HTMLAllCollection(8) [html, head, title, body, ul, li, li, li]
    ▶ anchors: HTMLCollection []
    ▶ applets: HTMLCollection []
      baseURI: "file:///C:/Users/Marcin/Desktop/example.html"
      bgColor: ""
    ▶ body: body
      characterSet: "windows-1252"
      charset: "windows-1252"
      childElementCount: 1
    ▶ childNodes: NodeList(2) [<!DOCTYPE html>, html]
    ▶ children: HTMLCollection [html]
      compatMode: "CSS1Compat"
      contentType: "text/html"
      cookie: ""
      currentScript: null
```

Figure 2.4 Properties of an 'html' object.

If the HTML document is not structured correctly, there will be no errors or warnings produced by the browser and it will attempt to parse the document regardless. Current browsers are cleverer about parsing and will try to complete the missing parts in the tree by assuming developer’s intent, but most of the time it will not succeed producing unexpected results. For example not including the end tag for the second list item of the unordered list from the previous example will not be a problem for a browser such as the current version of Google Chrome, but typing “</” instead of “</li>” for the closing tag will cause a misunderstanding. In this scenario the generated DOM tree will be faulty, mixing up two of the items as presented in figure 2.5.



```
▼<ul>
   <li style="color: red;">Item 1 in red</li>
  ▼<li style="font-style: italic;">
     "Item 2 in italic"
     <!--
         <li style="color: red; font-style: italic;"-->
     "Item 3 in both"
   </li>
 </ul>
```

Figure 2.5 Interpretation of a faulty DOM tree.

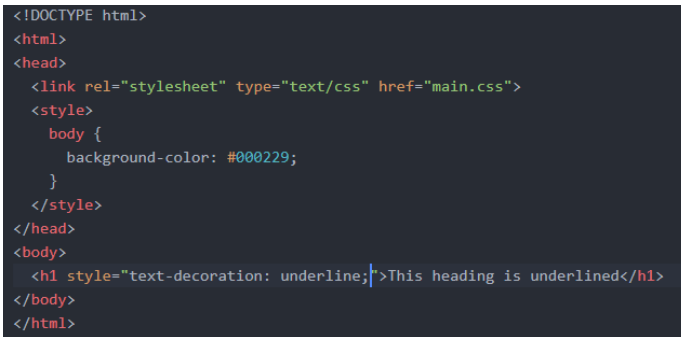
2.2 Cascading Style Sheets (CSS)

CSS is a style sheet language used for describing the presentation features of web pages written as a Hypertext Markup Language document[3]. It was first proposed by Håkon W. Lie in 1994[4]. At that time the HTML was pushed to its limits with things like “<font>” tags in order to offer the graphics designers the layout capabilities they needed[5]. This caused the actual HTML documents to raise their complexity, therefore making them harder to develop and maintain. The CSS solution resolved this problem by allowing the developers to separate the design from the semantic structure of web pages. This separation also improves the content accessibility and allows for more flexibility in the specification of the presentational features of the linked document.

2.2.1 Including CSS code in HTML

There are three different ways to include CSS code in HTML documents. The first way has already been briefly described in the previous section, ie. declaring the CSS inline for a specific element using the “style” attribute. This method is rarely useful as usually all the elements follow a certain design convention and if there was a need to alter this convention, all of the elements would have to be changed separately.

We can also put plain CSS code between the tags of a metadata HTML element “<style>”, which can be included in the “<head>” component of the document. The code included will then apply to all the contents of the document. The last method is to import a plain CSS file using another metadata element denoted by the “<link>” tag and passing the stylesheets’ URL as an attribute. With this approach, many HTML files can share the same styling convention contained in just a single stylesheet. Figure 2.6 presents a document using all of the above methods.



```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css" href="main.css">
  <style>
    body {
      background-color: #000229;
    }
  </style>
</head>
<body>
  <h1 style="text-decoration: underline;">This heading is underlined</h1>
</body>
</html>
```

Figure 2.6 Including CSS code in an HTML document.

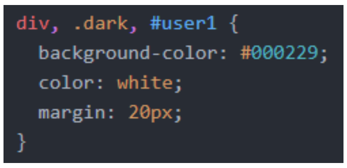
2.2.2 CSS syntax and usage

CSS has a very intuitive syntax using English keywords to specify the names of numerous style properties. A style sheet contains a list of rules. Each rule consists of one or more selectors, and a declaration block[6].

The selector allows the developer to select the target elements based on their properties, by declaring criteria that must be met by an element. The selector may specify things such as the elements’ type, the expected value of a particular attribute, and their placement relative to other elements in the DOM tree. Selectors may be combined in many ways to achieve great specificity and flexibility[7].

The declaration block simply lists the new values to be applied on specified properties for all of the target elements. The syntax for a single declaration is as quoted: “property_name: value;” and the whole block is contained between curly brackets. The complete list of properties as well as their possible values are specified in the CSS standard. Some properties can affect any type of element, while others may only be applied to particular subsets of them[8].

There are two attributes that are particularly useful when composing stylesheets, they are the “id” and “class” attributes. The id represents the unique identifier of an element within the document. By using the “#element_id” selector, we can create a rule for a specific element. On the other hand, the “.element_class” selector lets us set the style characteristics of all the elements sharing the given class or a set of classes. The Figure 2.7 shows an example CSS rule that will apply to all elements of type “<div>”, those with the class attribute set to “dark” and the element with its id equal to “user1”.



```
div, .dark, #user1 {
  background-color: #000229;
  color: white;
  margin: 20px;
}
```

Figure 2.7 CSS declaration block example.

2.3 Front-end frameworks

A framework is a standardized set of tools for solving a common type of problems, helping to produce solutions for new problems of a similar nature. In web development we can distinguish three types of frameworks that are used i.e. front-end frameworks provide commons structures for the presentation layer and some of the logic of a website on the client side, back-end frameworks let us manipulate the logic on the server side, and data layer frameworks are used to standardize data representation in databases[9].

Front-end frameworks typically consist of a package containing a collection of standardized HTML, CSS and JavaScript code, which can then be included in new projects. The most common features introduced by a front-end framework are:

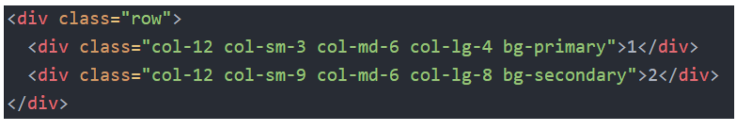
- CSS source code for creating a grid for placing elements which will adjust their position and size depending on the screen resolution allowing for compatibility for various devices from mobile phones to TV displays.
- Easy ways to create popular website elements which are more complex, for example a navigation bar.
- Set of style definitions for HTML elements allowing to maintain a similar theme and colour palette on the website.

- Compatibility for the most common browsers to ensure that a website built using the framework is exactly the same regardless of to the browser used to render it.

2.3.1 Bootstrap

Bootstrap is a free, open-source front-end framework which was originally created to be an internal tool at Twitter and is used by web developer teams to encourage consistency across their website[10]. Since it has been released for use outside of Twitter, it has become one of the most common front-end frameworks. Bootstrap is a perfect study case to explore how front-end frameworks can be used and what kind of validation requirements they can introduce.

One of the key features of Bootstrap is the grid system allowing to control the layout of the website based on the size of the display as mentioned in the previous paragraph. In order to use it we need to create a div element with class row and put div element with class col which can be treated as the individual cells of the grid. A row can display a maximum of 12 columns across its horizontal space, each getting the same fraction of space across the horizontal axis. If there are only two columns, each will take 50%[11]. Column class can be stricter by giving a number a number to it for example “col-3”, letting the column use only 25% of the width of the parent element. If the numbers add up to more than 3 in a row, the overflowed items will appear below the previous ones. We can also specify the size of the screen to which they should apply. There are five different thresholds used for the display sizes. The simple “col-1” class applies to extra small devices, “col-1-sm” applies to small devices. There are also classes for medium, large and extra large devices. Bootstrap allows for using column classes for multiple sizes in a single element. If a certain size class is not defined, one of the smaller defined sizes will be rendered. Figure 2.8 shows an example use of a grid system, showing that the ratios of the “col” elements are dependent on the display width.



```
<div class="row">
  <div class="col-12 col-sm-3 col-md-6 col-lg-4 bg-primary">1</div>
  <div class="col-12 col-sm-9 col-md-6 col-lg-8 bg-secondary">2</div>
</div>
```

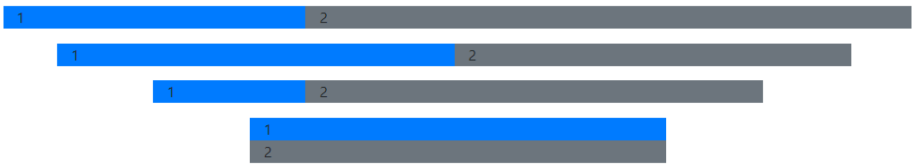
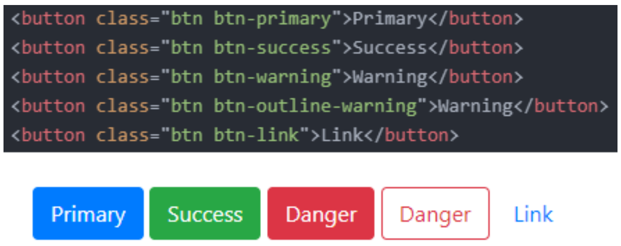


Figure 2.8 Bootstrap grid system usage example.

Many other components in Bootstrap have a similar requirement about the hierarchy of the types of the HTML elements together with their CSS classes used to produce them. For instance in order to create a valid card Bootstrap component, we need to put an element using “card-body” CSS class within the one using “card”, where both of them are of type “div”. As a result the use of CSS Bootstrap classes often imitates the actual type of the HTML element, which has similar requirements about the types of the parent and child elements that they can accept.

Bootstrap also features a set of styles for most the native HTML components. For example we can use a number of classes for the button element as presented on figure 2.9, showcasing the usage of some of the available available options.



```
<button class="btn btn-primary">Primary</button>
<button class="btn btn-success">Success</button>
<button class="btn btn-warning">Warning</button>
<button class="btn btn-outline-warning">Warning</button>
<button class="btn btn-link">Link</button>
```

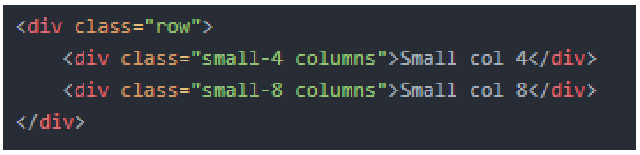
Figure 2.9 Bootstrap button class usage example.

2.3.2 Similarities and differences between frameworks

There exists a long list of front-end CSS frameworks other than Bootstrap I have chosen to examine the following ones focusing on the unique validation requirements they introduce to the HTML documents:

- Skeleton
- Foundation
- Pure
- Materialize
- Semantic-UI

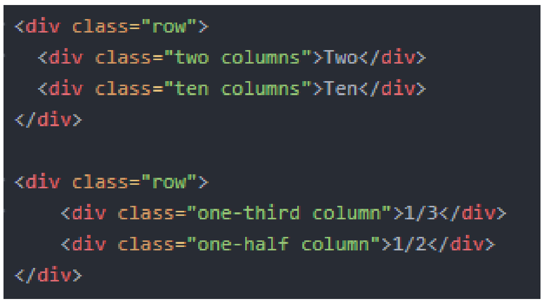
Although all of these frameworks apply very different designs, most of their features are implemented in a similar manner, requiring comparable type of validation rules, with the only difference being the names of the classes used in a particular framework. For example if we look at how to implement a grid component in Foundation (figure 2.10) we can see that it is very easy to translate this concept to Bootstrap where we could use “col-sm-4” instead.



```
<div class="row">
    <div class="small-4 columns">Small col 4</div>
    <div class="small-8 columns">Small col 8</div>
</div>
```

Figure 2.10 Implementation of grid system in Foundation.

There will always be special cases that will have to be taken into account, for example the grid system in Skeleton framework (figure 2.11) which uses full words instead of numbers when specifying the size of a column. It also allows it to be defined using a set of fractions where “one-third” is equivalent of “four”. I will have to ensure that my language is flexible enough to allow for establishing custom validation rules for such cases.



```
<div class="row">
  <div class="two columns">Two</div>
  <div class="ten columns">Ten</div>
</div>

<div class="row">
    <div class="one-third column">1/3</div>
    <div class="one-half column">1/2</div>
</div>
```

Figure 2.11 Implementation of grid system in Skeleton.

2.4 Linting

Linting is a process of running a program which can analyse given code, find and emphasize potential errors, bugs, and spacing mismanagements. The term itself originates from the first program carrying out the mentioned tasks. The program “Lint” program has been developed in 1994 and it was designed to validate C code[12].

2.4.1 ESLint

ESLint[13] is a modern, open source linting utility for JavaScript that can be embedded into Integrated Development Environments that support plugins. Thanks to that it can run a validation after every code alternation, giving the developer real time feedback. It also formats the parts of code to different colours, making it easier to distinguish different types of objects it represents, and also propose potential fixes to the discovered errors, that can auto generate missing pieces of code. ESLint has a large set of rules[14] that can be applied or omitted, depending on how strict we want to be about our coding style.

2.4.2 Bootlint

Bootlint is a HTML linter for Bootstrap projects checking for a range of common mistakes, with the main assumption being that the given web pages are already valid HTML5. All possible warnings and errors that can be raised by Bootlint are listed and well explained on a dedicated wiki page[15] created by the developers. The list includes 18 types of warnings and 52 types of errors, Bootlint includes a feature for excluding certain lint checks by using a "--disable" command followed by a set of warning/error codes to be ignored.

There are many ways to utilise Bootlint including commonly used JavaScript task runners (support for Grunt and Gulp), command line and running checks directly in the browser. If we choose to use it in a browser, all we need to do is set up a bookmark that executes a piece of JavaScript code specified in the URL block. This allows us to see the discovered problems directly in the browser console, meaning it can be easily applied to generated pages.

2.5 Domain-specific languages

A domain-specific language is a software language specialized for use in a particular application domain. DSLs are especially convenient for solving problems within clearly definable domains. DSLs are widely used across the industry, including languages for common domains, such as SQL used by many developers to create and query databases, but also languages used by private companies for their specific needs, for example to calculate salaries. The special-purpose nature of DSLs can effectively improve productivity and quality if used correctly, rather than using a general purpose language, which would require the building of all of the features that are already implemented in the DLS equivalent.

2.5.1 Language specification

A language specification for a given language L is the set of rules according to which the linguistic utterances of L are structured[16]. It allows us to efficiently describe the language, which is what we need in order to develop a DSL. Language specification consists of the following parts[16]:

- An abstract syntax model that specifies the abstract form of the language elements (figure 2.12).
- One or more concrete syntax models, each describing one of the concrete forms of the elements in the language (figure 2.13).
- Syntax mapping rules for each of the CSMs, defining how the concrete form of an element is transformed into its abstract form.
- A semantic description for each of the elements of the language.
- Optionally, interfaces which allow for easy translation to and from the elements written in a different language.

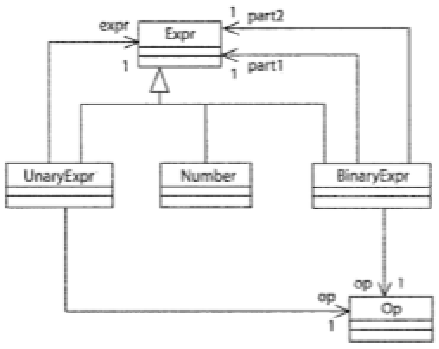
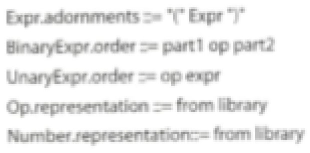


Figure 2.12 Abstract syntax model example. [16]



```
Expr.adornments ::= "(" Expr ")"
BinaryExpr.order ::= part1 op part2
UnaryExpr.order ::= op expr
Op.representation ::= from library
Number.representation::= from library
```

Figure 2.13 Concrete syntax model example. [16]

2.5.2 Metamodeling

Metamodeling is a formalisation method that can be used to define all the necessary elements of a software language specification ie. one metamodel to specify the abstract syntax, one metamodel to specify parts of the concrete syntax, and one to describe the semantics[16].

The crucial terms required to understand metamodeling are metamodel, model and instance. A model is a type graph similar to a UML class diagram combined with a set of constraints. An instance of a model M is a labeled graph that can be typed over the type graph of M and satisfies all the constraints in M's constraint set[16]. The node of an instance is properly typed if its label is equal to a node from the model graph. For an edge to be correctly typed it is required that the source and target labels are matching an edge from a model on top of its own label.

Finally a metamodel is a model used to specify a language, allowing it to produce models that apply the specified language, which are essentially instances of the metamodel.

2.5.3 DSL development tools

Developing a new domain specific language from scratch would involve reinventing the same ideas used in already existing ones. To avoid this situation, tools have already been developed which take a language specification described by the language engineer and produce the source code for the tool that are ready to use by the language users. Here is an example list of programming environments capable of such transformations:

- Eclipse Metamodeling Framework
- Jetbrains Meta Programming System
- Miclosoft DSL Tools
- Generic Modeling Environment
- International Domain Workbench
- MetaEdit+

2.6 Summary

This chapter covered all the basics of HTML, CSS, front-end frameworks, linting and domain specific languages which are the topics required for correct understanding of the problem introduced by project as well as its motivation.

3 Problem analysis

This chapter will focus on the motivation for the project as well as the potential challenges it may introduce. Such analysis will allow me to produce a list of requirements that the final implementation should fulfill.

3.1 Motivation

The number of CSS frontend frameworks has grown from the point where they were first introduced to the point where they can be treated as an extended domain of HTML use, which have their own validity criterias on top of the standard HTML syntax. Even if we are introducing our own CSS stylesheets, we often intend the selectors we have declared to be used in certain cases. An easy example would be to create a container class only for images, in which case we would like to avoid accidently including elements such as buttons within our container tags.

Surprisingly, even though there are many cases where using CSS inside HTML impacts the validity of the document, there are a lack of tools that would allow the developers to ensure that the website they have created or that has been generated with another application has the correct structure. The potential reason why CSS frameworks do not include convenient ways to check the structure of the document might be the fact that the website will always be rendered, regardless of the correctness in the framework domain. In my opinion this reason is not sufficient, as the result will still differ from the intentions of the developer. An additional linter will allow the developer to focus on the design rather than the syntax, making their work more productive.

Since all CSS frameworks are similar to some extent, it is possible to limit the validation rules to a specific domain. Developing a domain specific language for these rules would result in a very straightforward language allowing for simple definition of requirements that must be met by a HTML document using a certain framework. The language can then be interpreted by code generating software to produce a linting program making the process of creating HTML and CSS linters much quicker and accessible, which defines the main aim for this project.

3.2 Challenges

One of the main challenges that will be faced in to developing a solution to this problem is to keep the language simple and specific to the domain, but also complete enough to define some more complex constraints that may be useful for some frameworks. This is the most important matter that must be taken into account during the design process.

Creating a linting tool with use of the domain specific language will also need to be substantially easier and more intuitive than writing a linter from scratch in a standard programming. Otherwise there will be no reason for anyone to choose my implementation over the solutions which already exist.

3.3 Assumptions

The main assumption of this project is that the HTML documents on which the linting will be performed are already valid HTML documents. The linting will only concern the additional rules which are introduced by a particular CSS framework.

Another thing to keep in mind is the fact that the language will not cover concepts for every possible rule that might appear in a certain framework and not support validation for aspects which are very closely dependent on to the actual framework used. For this reason the system should allow for expandability of the program which performs the linting process.

3.4 Requirements

To ensure the overall quality of the implementation I have listed the requirements that the project should fulfill:

1. Abstract syntax capable of describing over 90% of the validation rules present in modern CSS frameworks.
2. Intuitive concrete syntax. Since the domain of the language is fairly narrow, the language can be self explanatory by including English phrases, allowing new users to understand the intentions of a previously defined validation rule just by reading them.
3. The language should not include redundant elements in order to keep it as simple as possible.
4. The system should include validation of the linters defined with use of the language in order to minimize the number of user errors.
5. Apply styles on certain keywords of the language when typed in the editor to make the language easier to read. For example using different colours could help to distinguish between certain types of selectors more easily.
6. The editor should include constraints which prevent the user from creating invalid framework descriptions.
7. A code generating component which creates a linting tool based on the validation rules described in the DSL.

8. The generated program should be accessible and easy to run with minimal setup required.
9. The system must be able to perform validation on HTML websites with the rules defined with respect to the domain specific language.
10. The linter included in the solution should allow for validation of dynamic websites and documents generated by other tools.
11. Allow for expandability of the linter in a general purpose language.
12. Documentation for the language listing all expressions of the language and the expected inputs for them.
13. A simple guide for the users should be included, listing and explaining all the necessary steps involved in the process of describing the validation rules for a framework of choice and generating a linter based on it, as well as the setup and usage instructions of the generated tools. The guide should also include instructions for potential expandability which may be needed for situations when the language was not sufficient to describe a certain rule required in a specific framework.

3.5 Summary

In this chapter we explored the motivation of the project and specified the requirements that the final implementation should be able to fulfill. Now that the problem is described in a formal manner we can move on to the design phase.

4 Design

This chapter will cover the design process of the elements required to create the domain specific language and propose the system architecture for the generated linter tools.

4.1 Linter representation

In order to produce a viable representation of a linter I have investigated the validation rules included in a previously mentioned existing solution for linting documents which utilize Bootstrap framework called Bootlint. This gave me an idea of what kind of validation rules are necessary for such a linter. I have grouped all the rules which seemed similar and ended up with eight distinct types of rules. I have decided to avoid the deprecated rules as well as the ones which are responsible for ensuring that the framework has been properly included as the method depends closely on the actual framework used. I will assume that it is already loaded before using the generated linter.

A linter can be represented by a named object which holds a collection of checks. The check itself should be an abstract class used as a basic blueprint shared by all types of checks. Every check will only concern elements which satisfy certain requirements. These requirements can be represented by a set of CSS selectors. It should also be labeled with a unique identificator useful for referencing. Every check can only raise an Error or a Warning upon violation which should also be specified for each check object. The actual types of checks will extend the Check class and introduce more specific parameters.

4.2 Abstract syntax

4.2.1 Metamodel

To represent my abstract syntax design I have created a metamodel diagram (figure 4.1) which shows the key components to be included in the language.

The root element of the language is the Linter element which can be given a name and specify the supported framework and its version. This information will be used to create the name of the generated program. Linter object can also hold two types of child components. The selectorDefinitions collections stores objects of SelectorDefinition type, which hold selectors that can be reused throughout the document. Selectors and SelectorDefinitions are a separate aspect of the language which will be explained with use of anther diagram (figure 4.2). Linter also stores a collection of various subtypes of Check concept which represent the actual validation rules of the framework.

The abstract concept Check shared across all subtypes holds an id number for later reference, a type of the violation message represented by a CheckType enumeration allowing for two string values: ‘Warning’ and ‘Error’. The last parameter is an optional custom message, allowing the user to overwrite the automatically generated violation message, which is generated appropriately to the subclass and the given parameters. Since all checks will apply to a certain set of HTML elements, the Check concept holds a collection of selectors, which will allow the linter to find these elements.

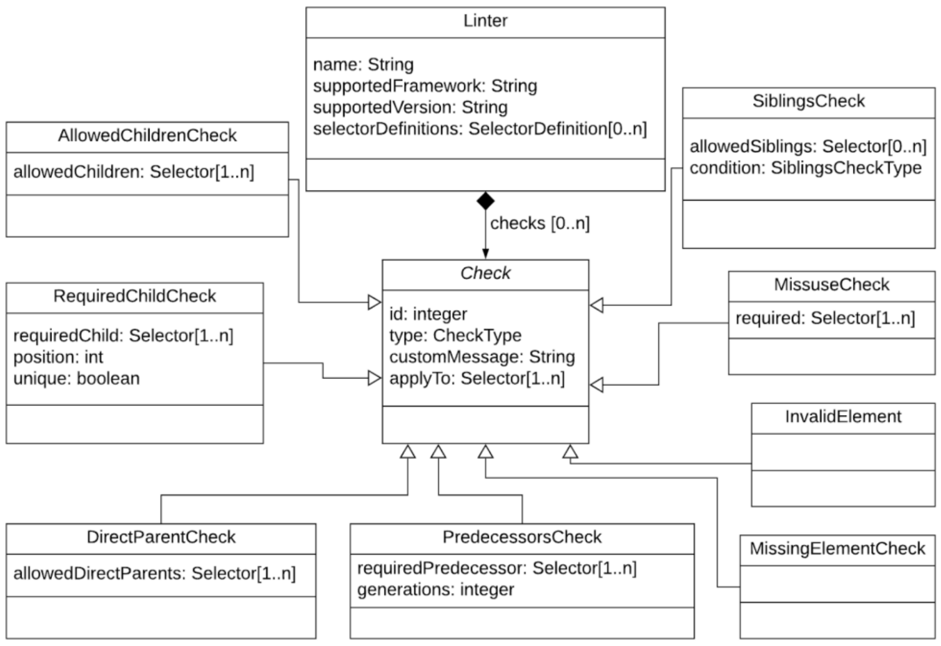


Figure 4.1 Abstract syntax metamodel diagram.

As we can see on the diagram, there are 8 subtypes of the abstract node Check. They will represent a specified group of validation rules. The design predicts the following check types:

- AllowedChildrenCheck - represents a validation rule which checks if all elements that it applies to have only children which match one of the given selectors.
- RequiredChildCheck - represents a rule which checks if all elements matching one of the selectors stored in the applyTo collection have a child matching a selector from a requiredChild list. The position of the child as well as allowing only one child of such type can also be specified by the class parameters. If position is set to 0 allow the child element to appear anywhere.

- PredecessorsCheck - describes a check for elements that must be placed within an element matching one of the given selectors. The ‘generations’ parameter allows to specify where exactly a certain predecessor should be expected. If it is set to 0 it should be ignored.
- DirectParentCheck - essentially the same as the previous check, but always concerns only one generation up.
- MissingElementCheck - represents a rule which checks if there is at least one element within the HTML document that satisfies one of the selectors given in the applyTo collection.
- InvalidElementCheck - this type of check should raise an error or a warning if there is an element matching any of the selectors in the applyTo collection.
- MisuseCheck - for this check to pass, all elements picked up by the applyTo selectors must also satisfy one of the required selectors from the other collection. This is useful if for instance a certain CSS class is designed to only be used on elements of type ‘div’.
- SiblingsCheck - represents 6 different checks related to siblings of the elements which match one of the selectors which can be specified by a SiblingCheckType. Thanks to that the user can check next immediate sibling, all next siblings or ensure that it has no next siblings. The other three options allow for the same operations, but regard previous siblings instead. If the user wants to check both, previous and next siblings, two separate checks are necessary.

The simple option to implement selectors into the system would be to allow the user to type an actual CSS selector and store it as a string. Although this would be sufficient, I have decided to implement my own representation which would not require the user to know the actual syntax of CSS selectors. Figure 4.2 shows all types of selector to be implemented in the solution, and describes the relation with the SelectorDefinition objects which can be child elements of the root component.

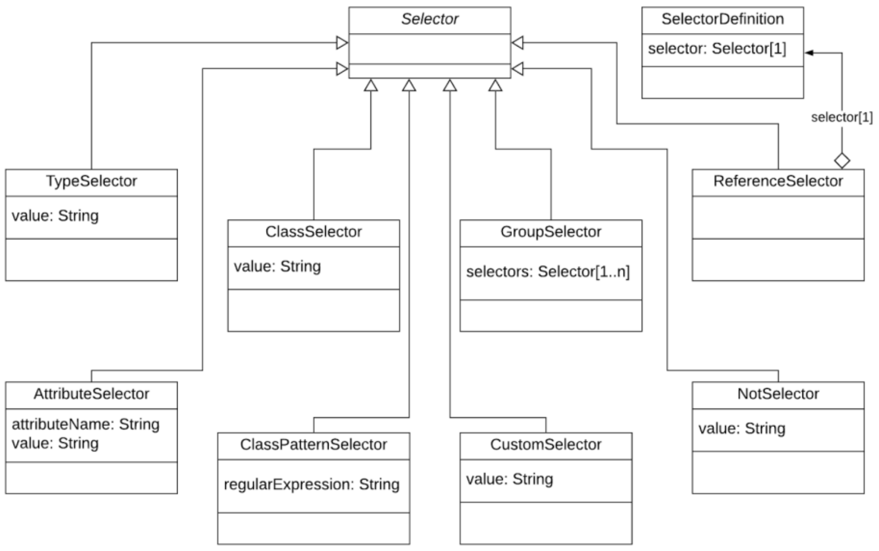


Figure 4.2 Selector sub-concepts.

Selector component itself is an abstract class which is extended by the following subconcepts:

- TypeSelector - represent a standard CSS selector for tags.
- ClassSelector - represents the CSS class selector. To translate it to a valid CSS selector, the text generation will only need to add a '.' character before the given string value.
- AttributeSelector - allows to represent the CSS attribute selector with use of the attribute name and the actual value expected.
- ClassPatternSelector - allows to select elements which have a class matching the given regular expression. This is helpful for cases such as the column elements in Bootstrap which can use many different classes which match a certain pattern.
- GroupSelector - allows to merge multiple selectors together, meaning that an element must obey all requirements specified inside it to be selected by the group selector.
- CustomSelector - a custom selector will allow the users which already know how CSS works to write their own selectors.
- ReferenceSelector - using a reference selector will point at a predefined selector and put its definition in a desired location.
- NotSelector - negates the given selector.

4.2.2 Constraints

The metamodel should allow me to build the structure of the language, but it cannot describe all constraints of the model. In order to complete the definition of the abstract syntax I have listed additional constraints that should limit the concepts of the language based on the assumptions concerning the domain. The design predicts the following constraints:

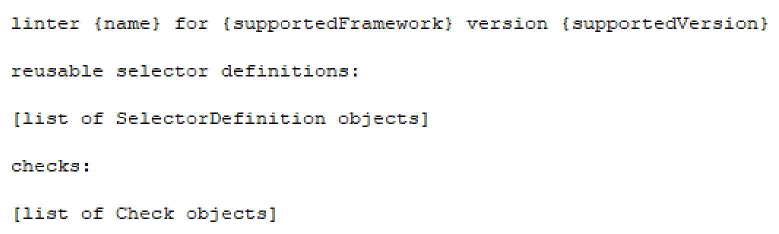
1. Every check id must be unique for correct referencing.
2. Every selector definition must have a unique name.
3. All selectors with a string value should not allow the user to input any characters that cannot be a part of any CSS selectors, as this might result in later crashes of the generated linter.
4. The GroupSelector should only allow for a single TypeSelector to be included as it works as an “and” operator, and there are no HTML elements which can be of two types at the same time. Since the selector can include other selectors which may contain their own child selectors, the constraint should also ensure that the resulting tree structure only contains at most one type selector leaf.
5. The NotSelector should never have a direct child of the same type, as this would only add redundancy to the code.
6. Selector definitions should not contain references to themselves as this would result in a loop behaviour during the generation of the linter. Since the loops can be created indirectly, the constraint must trace the branches of the tree structure.
7. SiblingsCheck should not specify any selectors in the allowed siblings collection if either “no next” or “no previous” siblings option is selected. Also it should expect at least one selector if any of the remaining options is chosen.
8. The position parameter of the RequiredChildCheck and the generations parameter of the PredecessorsCheck should not allow negative integers.

4.3 Concrete syntax

To make the concepts presented in the abstract syntax usable, we need a way to represent them. This will be specified by the concrete syntax which maps text input to concepts and their parameters.

I will aim to produce a syntax which can be read and understood like a normal document written in English. This might result in including redundant words, but the editor should allow the user to create the basic structure of a concept by writing an alias, only leaving the gaps for the actual parameters to be filled with the desired inputs.

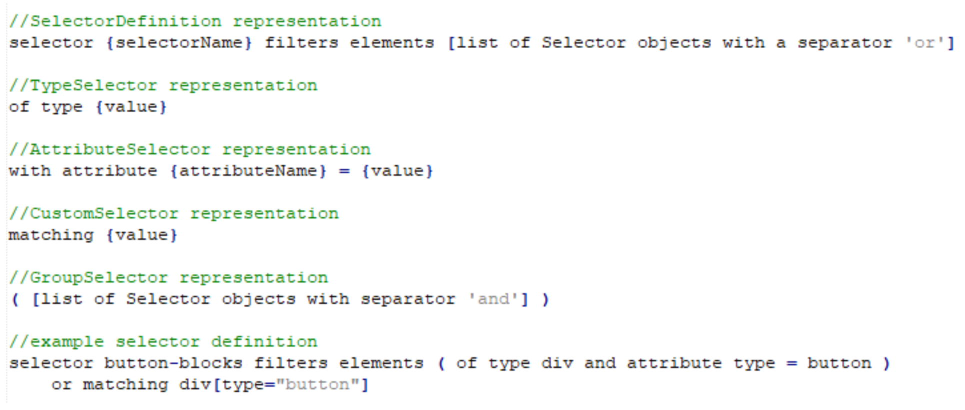
The design of the Linter concept representation which is the root component of the metamodel can be seen on figure 4.3. All text outside brackets is part of the syntax where as the text inside curly brackets represents the space to define parameters. The square brackets show where the body of relevant child concepts should appear.



```
linter {name} for {supportedFramework} version {supportedVersion}

reusable selector definitions:

[list of SelectorDefinition objects]

checks:

[list of Check objects]
```

Figure 4.3 Linter concept representation.

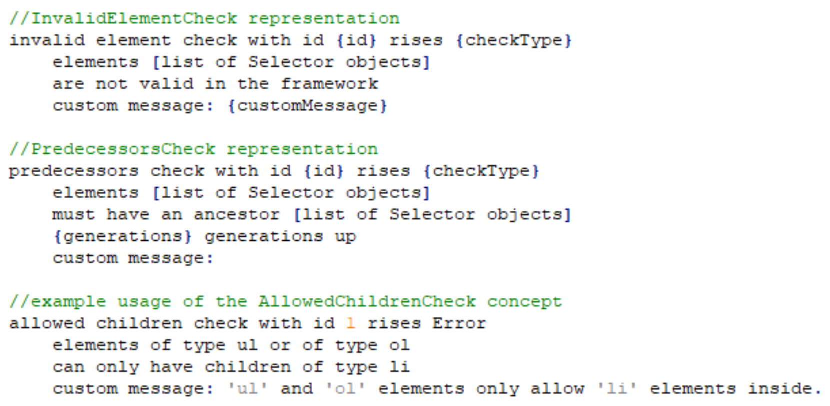
Concrete syntax for selector definitions and some of the selector concepts is presented in figure 4.4. Every collection of selectors will be separated by an ‘or’ word, with the only exception being the list inside a group selector which separates the selectors with an ‘and’. Every selector should be self descriptive. As we can see in the example, putting selectors together works well and even holds the grammar rules of the English language. If needed the custom selector can be used, which can replace any combination of the more basic selectors as we can see in the example where the group selector is essentially the same as the custom selector.



```
//SelectorDefinition representation
selector {selectorName} filters elements [list of Selector objects with a separator 'or']

//TypeSelector representation
of type {value}

//AttributeSelector representation
with attribute {attributeName} = {value}

//CustomSelector representation
matching {value}

//GroupSelector representation
( [list of Selector objects with separator 'and'] )

//example selector definition
selector button-blocks filters elements ( of type div and attribute type = button )
    or matching div[type="button"]
```

Figure 4.4 Concrete syntax design Selector concepts.

The crucial concepts of the language are the subtypes of class Check. The proposition for the representations of two checks can be seen on figure 4.5. Every check begins with a declaration of the check class used. The common elements of the checks such as the placement of the id, check type and the custom message are kept the same for all check to keep the concepts close to the abstract Check concept. The scope of every validation rule should be clear to anyone reading the check declaration which is the case in the example of potential usage of the concept for checking the allowed children of an element.



```
//InvalidElementCheck representation
invalid element check with id {id} rises {checkType}
    elements [list of Selector objects]
    are not valid in the framework
    custom message: {customMessage}

//PredecessorsCheck representation
predecessors check with id {id} rises {checkType}
    elements [list of Selector objects]
    must have an ancestor [list of Selector objects]
    {generations} generations up
    custom message:

//example usage of the AllowedChildrenCheck concept
allowed children check with id 1 rises Error
    elements of type ul or of type ol
    can only have children of type li
    custom message: 'ul' and 'ol' elements only allow 'li' elements inside.
```

Figure 4.5 Concrete syntax propositions for checks.

4.4 Editor

One of the crucial elements of the project is the editor which will allow the user to define linters using the proposed language. The editor should guide the user through what kind of concepts can be used at a certain place and automate the process of inserting long parts of the concrete syntax which are fairly common in the design due to the self-explanatory nature.

The editor should also format certain parts of the syntax through use of style attributes such as the colour and weight of the text to make the language more readable. Since the colour scheme is not that important I will ignore it at this stage of the project.

4.5 Generated Linter

Based on the requirements, the language must be translated to code that can be run by a computer. For this purpose I have decided to generate a JavaScript application that can be run by any browser, making it perfect for this system. The setup process will require the user to set up a bookmark that will run a function from the script. Changing the options of the linter will be available through arguments that will be passed in the function call that can be altered.

The program will include function definitions for each type of the validation rules. The actual checks will be translated to function calls with parameters specified in the model. The collections of selectors will be translated to arrays of native CSS selectors which can easily filter the elements of interests by using JavaScript.If a rule is violated, the program will print warnings and errors in the console of the browser. The application should also produce an alert with a summary of the check process.

The user should also be able disable some violation rules. To support this, the main function will include a parameter which takes an array of integers. If a check is present in this array, it will be omitted. The program will also allow to choose between grouping the violations by the check and print all cases which failed the check in one collection or to print violations separately for each element which caused problems. The second approach would also allow for a single iteration through the DOM tree, as the applyTo selectors will only have to be checked against a single element which is under investigation at a given moment.

To group these design ideas I have written a piece of pseudo-code (Appendix A) representing the main structure of the program, which should guide me through the process of implementation of the text generator.

4.6 Summary

Now that the key components of the project have been designed, they can be implemented. Due to the agile methodology, the design might undergo slight modifications during the implementation process.

5 Implementation

In this section I will present my solution and cover the crucial parts of the implementation process. Firstly, I will introduce the software used and explain the reasoning behind choosing it over other options. Then I will describe the details of individual components of the language, editor and the text generator, explaining how they were created.

5.1 Meta Programming System (MPS)

As previously mentioned in the literature review (report part 2.5.3) MPS is one of the development tools for domain specific languages that is currently available on the market. It allows for easy development of all the necessary components of a domain specific language as well as the tools which can be used by the language user to produce models that utilise it and generate executable code based on those models.

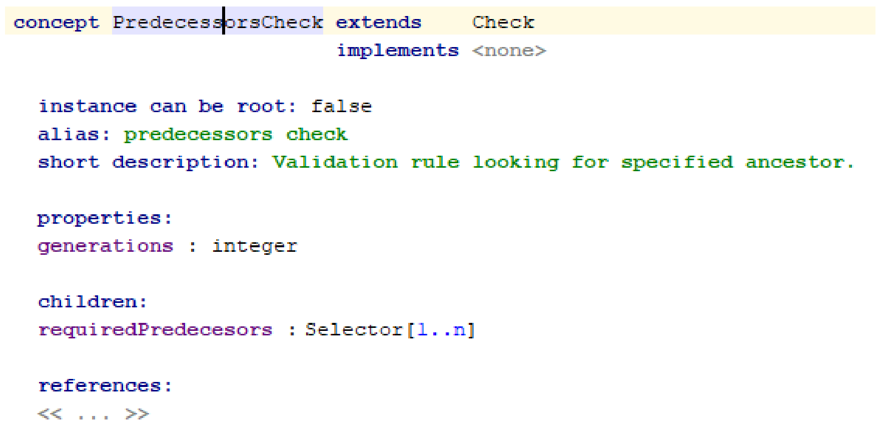
I have also taken the Epsilone Metamodeling Framework under consideration. It has similar features and a better text generator component, but it does not offer a facility for designing editors that would be fit for the purpose of this project. It supports a tree editor and a graphical editor, whereas MPS allows the developer to produce customizable text editors, which is required to implement the concrete syntax presented in the design section.

The language that I have developed in MPS is called LinterDSL and it contains all the components which are required to use it, including the language model, constraints, editor and the code generation feature. All the user has to do to start using LinterDSL is download MPS and create a new solution which includes my language in its dependencies.

5.2 Language structure

To develop an abstract syntax of the language I have created a set of concepts. Every concept in MPS extends the BaseConcept class which allows specification of the parameters, child elements and references to other objects included in the model. Each of these elements can be named and restricted with an expected type for the input. Children and reference lists can also be restricted with an expression which specifies the allowed size of the collection.

Concepts can also be given an alias and a short description. The alias allows the editor to refer to the concepts with a name which is easier to understand. This allows it to be presented to the user instead of the actual concept name and the description provides a short string explaining the purpose of the concept.



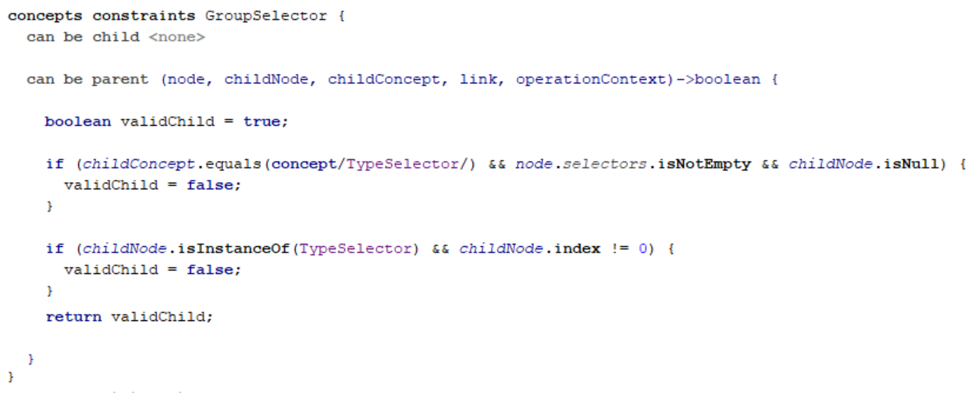
```
concept PredecessorsCheck extends    Check
                          implements <none>

  instance can be root: false
  alias: predecessors check
  short description: Validation rule looking for specified ancestor.

  properties:
  generations : integer

  children:
  requiredPredecesors : Selector[1..n]

  references:
  << ... >>
```

Figure 5.1 PredecessorCheck concept implementation in MPS.

To create the language structure I have strictly followed the design diagrams included in the previous chapter (figures 4.1 and 4.2). To present an example of an implemented concept I have included an MPS representation of the PredecessorsCheck in figure 5.1. For full specification of the structure refer to the language documentation (Appendix B).

5.3 Constraints

MPS allows for applying additional constraints to the concepts, which will inform the user about potential errors that they may have introduced in the editor. I have implemented all of the constraints mentioned in report part 4.2.2 using the features provided by MPS. To present an example of an implemented constraint I have included code for constraint which ensures that a group has at most one TypeSelector (constraint #4) in figure 5.2.



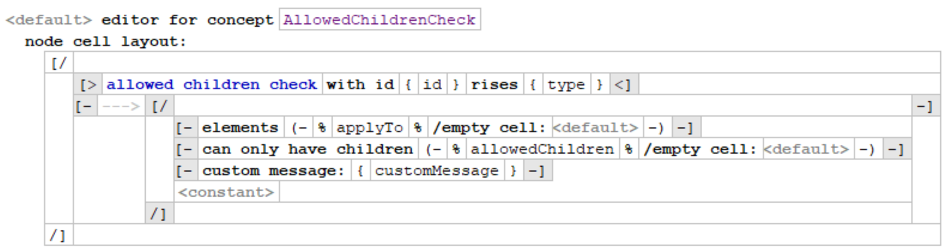
```
concepts constraints GroupSelector {
  can be child <none>

  can be parent (node, childNode, childConcept, link, operationContext)->boolean {

    boolean validChild = true;

    if (childConcept.equals(concept/TypeSelector/) && node.selectors.isNotEmpty && childNode.isNull) {
      validChild = false;
    }

    if (childNode.isInstanceOf(TypeSelector) && childNode.index != 0) {
      validChild = false;
    }
    return validChild;

  }
}
```

Figure 5.2 GroupSelector constraint implementation.

5.4 Editor

Creating an editor in MPS links to implementing the actual concrete syntax of the language. Each concept is given its representation to be used by the editor. The definition of an editor component for a concept is a template which contains string elements that are part of the language syntax and gaps for the parameters, child elements and references. The syntax introduced in the editor templates aims to sound like a typical English sentence.

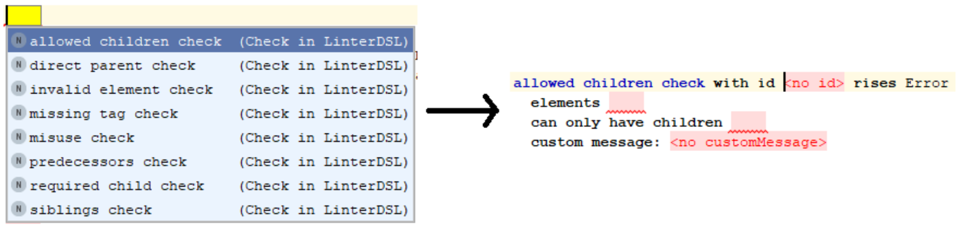
To make the linter definitions more readable I have applied colours to relevant parts of the syntax. Every check type is marked with a blue colour to highlight the most important part of the fixed syntax, and each selector type is given its own colour, so they can be easily distinguished.Figure 5.3 shows theMPS editortemplate for the allowed children check.



```
<default> editor for concept AllowedChildrenCheck
  node cell layout:
    [/
      [> allowed children check with id { id } rises { type } <]
      [- ---> [/                                                              -]
                 [- elements (- % applyTo % /empty cell: <default> -) -]
                 [- can only have children (- % allowedChildren % /empty cell: <default> -) -]
                 [- custom message: { customMessage } -]
                 <constant>
               /]
    /]
```

Figure 5.3 AllowedChildrenCheck concept template.

It would be very tedious for the user to have to type all the key words included in the syntax template and also make sure that there are no spelling mistakes in the syntax. To avoid this situation, in order to create an object of a desired concept the user can either type in its alias or use the Ctrl + Spacebar shortcut to display all valid concepts at a particular spot in the document and choose it from the list. Both options will result in printing all fixed parts of the template in the editor and only leave empty spaces for the editable inputs. We can see this behaviour on figure 5.4.



```
allowed children check      (Check in LinterDSL)
direct parent check         (Check in LinterDSL)
invalid element check       (Check in LinterDSL)
missing tag check           (Check in LinterDSL)
misuse check                (Check in LinterDSL)
predecessors check          (Check in LinterDSL)
required child check        (Check in LinterDSL)
siblings check              (Check in LinterDSL)
```

```
allowed children check with id <no id> rises Error
  elements
  can only have children
  custom message: <no customMessage>
```

Figure 5.4 Creating an object in the editor.

Only the valid concepts will appear on the list and using an alias which is not allowed at a certain place will not result in an object creation. This limits the possibility of human error by a fair amount. Any parameter inputs that violate the constraints are picked up by the editor and highlighted with red.

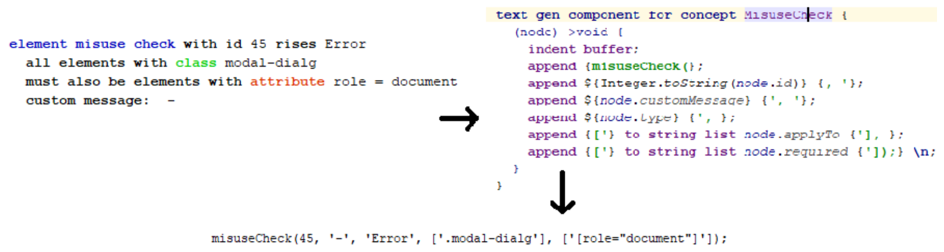
The concrete syntax for the Linter concept, which is the root component of the language is matching the design from figure 4.3. The representation of selector definitions also matches the initial design. The templates of the individual check types aim to explain the purpose of the check within the fixed syntax. We can see an example on the figure 5.2.

The templates of the individual selectors aim to inform the user about the type of the current selector, but also keep the flow of the sentence. For example the syntax for the class selector is ‘with class {value}’. If we combine it with the second line of the AllowedChildrenCheck template (figure 5.2) we will get ‘elements with class {value}’ which is grammatically correct. For the same reason I have implemented an ‘or’ separator for every collection of selectors. The group selector is a special case and its representation looks as follows ‘([list of selectors with separator= “and”])’. To see an example of a more complete result of using the editor please look at appendix E.

5.5 Generated linter

In order to create an executable linter, the model created using my language must be transformed to a general purpose programming language. To enable this translation I have used the TextGen feature included in MPS. It allows for creating templates which fill the gaps with the values of properties describing the objects of a model.

I followed the design and aimed to translate the checks to JavaScript function calls (figure 5.5) and the selectors to strings which represent their equivalent CSS selectors. For example to translate a class object to a CSS class selector, the text generator has to append a ‘.’ sign before the string value and to represent the group selector, the generator concatenates the selectors string representations together. The only selector which caused more problems to implement was the ClassPattern selector. To support it I have used a ‘regex()’ selector extension for jQuery[17] which is available online[18]. It allows the user to propose regular expressions that will file elements with matching classes.



```
element misuse check with id 45 rises Error
  all elements with class modal-dialg
  must also be elements with attribute role = document
  custom message:  -
```

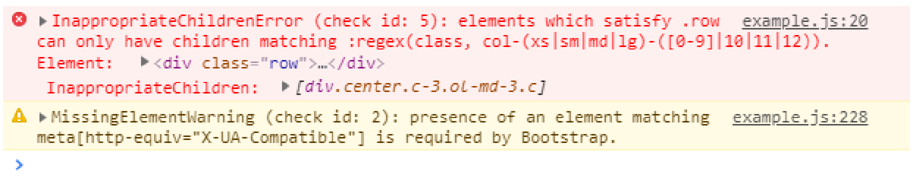
```
text gen component for concept MisuseCheck {
  (node) >void {
    indent buffer;
    append {misuseCheck(};
    append ${Integer.toString(node.id)} {, '};
    append ${node.customMessage} {', '};
    append ${node.type} {', };
    append {['} to string list node.applyTo {'], };
    append {['} to string list node.required {']);} \n;
  }
}
```

```
misuseCheck(45, '-', 'Error', ['.modal-dialg'], ['[role="document"]']);
```

Figure 5.5 MisuseCheck text generation process.

The functions that execute the actual validation rules are part of the root element table. To detect the violations I am using jQuery features, which allows me to easily filter the elements that I am interested in. All checks follow the structure proposed n the design section (appendix A) allowing the use of the single iteration mode and disabling a list of checks. To generate the linter, the user has to use the build option on the model solution. This will result in creating a JavaScript file named the same as the root element of the model. If the user wishes to add their own validation rules to the linter. They can do so by extending the generated code.

To run the generated linter the user has to create a bookmark which includes a function call of the main function of the program. For more specific instructions please refer to appendix C which explains the setup process in more details. Upon running the linter, the user is notified with an alert in the browser and instructed to open the console. If any violations are encountered in the currently opened website, they will be printed in the browser console. We can see an example of it in the figure 5.6.



```
▶InappropriateChildrenError (check id: 5): elements which satisfy .row   example.js:20
can only have children matching :regex(class, col-(xs|sm|md|lg)-([0-9]|10|11|12)).
Element:   ▶<div class="row">…</div>
 InappropriateChildren:  ▶[div.center.c-3.ol-md-3.c]
▶MissingElementWarning (check id: 2): presence of an element matching   example.js:228
meta[http-equiv="X-UA-Compatible"] is required by Bootstrap.
>
```

Figure 5.6 Example violation messages.

5.6 Summary

In this chapter I have presented the final state of my solution and explained how it integrates its components together. In addition to the functional system I have also created a full documentation of the language (appendix B) as well as a detailed user guide explaining how to create a linter model and also how to generate and use the JavaScript program based on it. The source code of my implementation is available in an online repository[19].

6 Evaluation

This chapter will evaluate the final implementation of the project. I will assess the solution against the requirements listed in part 3.4 with appropriate tests and present an example use case for the domain specific language proposed in the project.

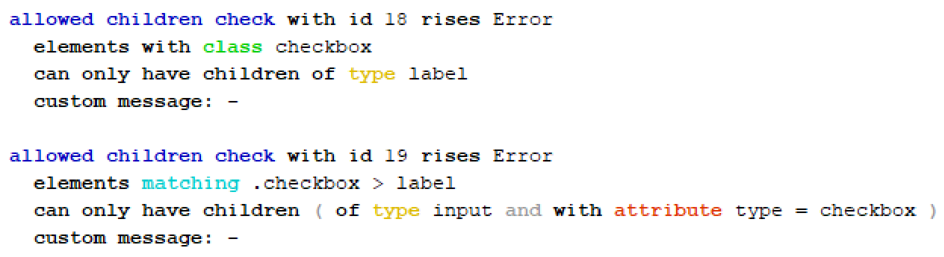
6.1 Language domain coverage

To evaluate the completeness of the language within the domain I have decided to implement an existing linter with use of the LinterDSL language. I will then count the validation rules included in the existing tool which fall within the assumed domain and see how many of them can actually be represented with my language. This will allow me to calculate the percentage coverage of required validation rules for a specific framework.

For this purpose I have chosen to redefine the newest version of Bootlint for Bootstrap 4 as it can work similarly to the scripts generated with use of my system and display the violation messages in the browser console. After analysis of the Bootlint Wiki page I have counted 52 validation rules which fall within the assumed domain scope and 8 rules which are too closely related to Bootstrap itself to be taken under consideration.

I have managed to translate 50 validation rules out of the investigated 52 rules meaning that the overall Bootstrap validation coverage is over 96%. A snippet of the Bootlint redefinition can be seen in appendix E. I have also provided a mapping of the rules from the original Bootlint to the one defined in LinterDSL (appendix D). Some of the more complex rules had to be translated with use of multiple check concepts, but even though they had to be split, they are just as effective as the original rules. Every subtype of the check concept has been used in the Bootlint definition, meaning that there are no redundant elements in the language.

Even though the coverage is satisfactory for 5 rules there was no possibility to express them without the use of the CustomSelector concept. For example the rule E017 from Bootlint requires that elements with ‘checkbox’ class have child elements of type ‘label’ which contain ‘input’ elements within them. This requirement can be described with use of two RequiredChildCheck concepts as presented on figure 6.1, but it requires the user to utilise plain CSS selectors, which I have aimed to avoid by introducing my own selector concepts. They should be an option rather than the only possible way to implement a rule.



```
allowed children check with id 18 rises Error
  elements with class checkbox
  can only have children of type label
  custom message: -

allowed children check with id 19 rises Error
  elements matching .checkbox > label
  can only have children ( of type input and with attribute type = checkbox )
  custom message: -
```

Figure 6.1 Bootlint E017 rule translated to LinterDSL.

6.2 Syntax

The syntax fulfils the requirement number 2 from the list included in part 3.4 as the validation rules can be read as a typical English sentence. The purpose of each of the check types is also explained within the syntax. Unfortunately I did not gather any feedback from other people about the intuitiveness of the language hence I cannot provide objective evaluation.

6.3 Editor

The requirements regarding the quality of the editor have also been fulfilled. The editor is following the predefined constraints for the concepts allowing only valid objects to be created at a certain place of the model, minimizing the possibility of creating invalid models. The editor also informs the user about any invalid values of parameters by highlighting the errors, preventing them from creating faulty linting tools.

The editor also introduces colours to appropriate parts of the syntax to focus the user on the most important aspects of the designed model. This fulfills the requirement number 5.

6.4 Generated linter

The system is capable of generating executable code based on validation rules proposed in my language, fulfilling the requirement number 7. All that is involved in the setup process is creating one bookmark which runs the script, making the generated linter easy to use. The main function accepts two arguments allowing the user to ignore some checks and choose between two modes which describe how the violation messages should be printed, adding extra functionalities to the system. Since the linter can be run in a browser it will check the current state of the opened website allowing for validating dynamic pages. The generated JavaScript code can always be extended with additional validation rules introduced by another developer. All of these characteristics combined result in meeting requirements 7-11.

To assess the functionality of the generated linter I have created a test HTML document which violates all 50 rules included in the redefined version of Bootlint described in LinterDSL. The purpose of this website is to run both the original and the generated linters on it and see if the generated version manages to pick up all the violations found by the original. The test was successful as the generated linter discovered all errors included in the document. The console log from both runs in provided in appendix F.

6.5 User guidance

The solution includes a complete documentation of the LinterDSL language (appendix B) as well as a user guide explaining how to describe, generate and use the generated linters (appendix C). These resources fulfill the requirements number 12 and 13.

6.6 Summary

This section focused on the evaluation of the components included in the project. The project can be considered an overall success, but the evaluation managed to reveal areas in which the language could be improved. The major issue with my evaluation method is the lack of attempt to gather opinions about the language from other people. I have assessed the intuitiveness of the editor based on my own experience with linters.

7 Conclusion

This project aimed to design and implement a domain specific language which allows to define linting tools for HTML documents which utilise modern CSS frameworks and evaluate the coverage of the domain. The secondary goal was to allow for generating executable code performing the linting process based on the description and provide proof for its functionality.

The project involved deep analysis of a specific domain concerning the CSS frameworks in order to propose a metamodel which can effectively represent all the concepts that may be encountered when creating validation tools for such frameworks. The design proposed detailed plans for each of the necessary components of the implemented solution.

The chosen approach for the implementation of the language was to use a meta programming system (MPS) from JetBrains which provides all the necessary tools for developing custom domain specific languages. By using already existing software I was able to avoid reinventing the wheel and focus on the quality of the language itself. Once the linter is described with use of my language it can be transformed to executable JavaScript code that can be run by a browser.

Both the completeness of the language as well as the functionality of the generated tool have been assessed with satisfactory results proving that the domain specific language approach is valid for the purpose of representing HTML and CSS linters.

The project can be considered a success as all of the requirements established after the problem analysis have been met by the final implementation.

7.1 Further work

As previously mentioned in the evaluation section, the language has some faulty elements that could be addressed in future versions of my implementation. There are not enough selectors included in the current version of LinterDSL forcing the user to use the custom selector concept which utilises plain CSS syntax. To improve this the set of supported selectors should be increased or the whole idea of replacing the CSS selectors should be dropped.

Another way to enhance the quality of my project would be to gather opinions about the language from other developers in order to evaluate the intuitiveness of the language. We could then alter the features which they consider problematic and maybe add features that are missing.

The solution could also be integrated with a tool which validates the correctness of the document on the standard HTML domain, requiring to use only one application to check both, the standard HTML structure and the validation rules introduced by the framework.

Once these improvements are implemented, the project would be ready to use by anyone who wants to produce a linter for their framework with ease and hopefully result in an increase of the linters to frameworks ratio in the industry.

Appendix A

Generated linter code design presented as pseudo-code.

```
exampleCheckFunction(id, customMessage, type, applyTo ...) {
    if id not in disabled {
        if singleIterationMode {
            perform check on investigatedElement
            if violated {
                if customMessage not equals '-' {
                    message = customMessage
                } else {
                    message = generateMessageForExamCheck
                }

                print error/warning message with the investigatedElement
            }
        } else {
            misusedElements = []
            for element in all elements selected by applyTo selectors {
                perform check on element

                if violated {
                    add element to misusedElements
                }
            }
            if misusedElements length > 0 {
                if customMessage not equals '-' {
                    message = customMessage
                } else {
                    message = generateMessageForExamCheck
                }
                print error/warning message with misusedElements
            }
        }
    }
}
runChecks() {
    //Check concepts translated to function calls.
    exampleCheckFunction(1, '-', 'Error', ['div', '.col'])
}
main(disabled, singleIterationMode) {
    if singleIterationMode {
        for element in allDomElements {
            investigetedElement = element
            runChecks();
        }
    } else {
        runChecks();
    }
}
```

Appendix B

LinterDSL documentation.

1. Linter (root element)

The Linter object is the main element of any LinterDSL document. It holds the collections of checks and reusable selector definitions.

Syntax :

```
linter <no name> for <no supportedFramework> version <no supportedVersion>

reusable selector definitions:

  << ... >>

validation checks:

  << ... >>
```

Parameters:

- name (string): name value of the linter object. The generated file will be named the same as the linter object.
- supportedFramerwork (string): place to specify the framework name on which the linter will be used.
- supportedVersion (string): version of the supported framework. Both parameters (supportedFramework and supportedVersion) will be placed in a comment of the generated linter for informational purposes.

Collections:

- selectorDefinitions (Selector[0..n]): holds Selector objects that can be used throughout the documents.
- checks (Check[0..n]): collection of validation rules for the framework.

2. Check

Check is an abstract concept of the language which holds the common elements of its subtypes.

Syntax part common to all subtypes:

```
<alias of the subtype> check with id <no id> raises <no type>
  elements << ... >>
  <place for additional parameters and collections depending on the subtype>
  custom message:  <no customMessage>
```

Parameters common to all subtypes:

- id (integer): unique identifier of the check.
- type (CheckType): enumeration which can be set to one of the following: ‘Error’, ‘Warning’.
- customMessage (string): custom message can override the automatically generated message upon violation. To use the automatic message set the value to the ‘-‘ sign.
- applyTo (Selector[1..n]): describes the set of elements to which the check will be applied.

2.1 AllowedChildrenCheck

Ensures that all children of elements matching one of the selectors specified in the applyTo collection match one of the selectors specified in the allowedChildren collection.

Alias: allowed children check

Syntax:

```
allowed children check with id <no id> raises Error
  elements <applyTo>
  can only have children <allowedChildren>
  custom message: <no customMessage>
```

Parameters:

- allowedChildren (Selector[1..n]): specifies the set of allowed children for the investigated element.

Default violation message:

- InappropriateChildren{type} (check id: {id}): elements which satisfy {applyTo} can only have children matching {allowedChildren}. Elements with inappropriate children: [list of invalid elements]

2.2 DirectParentCheck

Ensures that an element is placed within a right parent.

Alias: direct parent check

Syntax:

```
direct parent check with id <no id> raises <no type>
  elements <applyTo>
  must have a parent <allowedParents>
  custom message: <no customMessage>
```

Parameters:

- allowedParents (Selector[1..n]): specifies the set of allowed direct parents for the investigated element.

Default violation message:

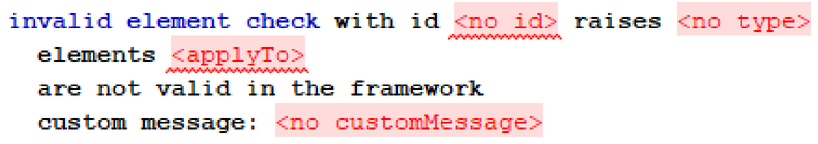
- DirectParent{type} (check id: {id}): elements which satisfy {applyTo} can only be direct children of elements matching: {allowedParents}. Elements with inappropriate parents: [list of invalid elements]

2.3 InvalidElementCheck

Raises a Warning/Error if it finds elements matching any selectors given in the {applyTo} set.

Alias: invalid element check

Syntax:



```
invalid element check with id <no id> raises <no type>
  elements <applyTo>
  are not valid in the framework
  custom message: <no customMessage>
```

Default violation message:

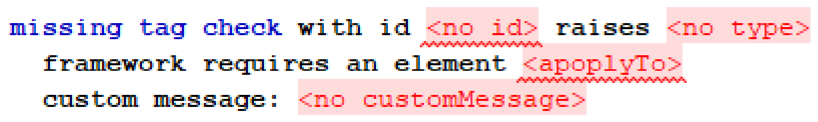
- InvalidElement{type} (check id: {id}): elements matching {applyTo} are not valid in {Linter.supportedFramework}. InvalidElements: [list of invalid elements]

2.4 MissingTagCheck

Raises a Warning/Error if no elements matching any selectors specified by the {applyTo} set are found within the document.

Alias: missing tag check

Syntax:



```
missing tag check with id <no id> raises <no type>
  framework requires an element <apoplyTo>
  custom message: <no customMessage>
```

Default violation message:

- MissingElement{type} (check id: {id}): presence of an element matching {applyTo} is required by {Linter.supportedFramework}.

2.5 MisuseCheck

Applies additional requirements for the elements picked up by the {applyTo} selectors by listing another set of selectors that must be satisfied by them.

Alias: element misuse check

Syntax:

```
element misuse check with id <no id> raises <no type>
  elements <applyTo>
  must also be elements <required>
  custom message:  <no customMessage>
```

Parameters:

- required (Selector[1..n]): specifies a set of additional requirements for the selected elements.

Default violation message:

- ElementMissuse{type} (check id: {id}): elements which satisfy {applyTo} must also match {required}. Misused elements: [list of invalid elements]

2.6 PredecessorsCheck

Ensures that all elements matching one of the {applyTo} selectors are placed as a descendant of an element specified by another set of selectors.

Alias: predecessors check

Syntax:

```
predecessors check with id <no id> raises <no type>
  elements <applyTo>
  must have an ancestor <requiredPredecessor>
  <no generations> generations up
  custom message: <no customMessage>
```

Parameters:

- requiredPredecessor (Selector[1..n]): specifies the set of selectors that must be matched the required ancestor.
- generations (integer): specifies the how far apart in terms of generations the elements should be to the required ancestors. If the value is set to ‘0’ all previous generations will apply.

Default violation message:

- Predecessor{type} (check id: {id}): elements which satisfy {applyTo} must be descendants of an element matching: {requiredPredecessor}. Misplaced elements: [list of invalid elements]

2.7 RequiredChildCheck

Ensures that all elements matching one of the {applyTo} selectors are have a required child element specified by another collection of selectors.

Alias: required child check

Syntax:

```
required child check with id <no id> raises <no type>
  elements <applyTo>
  must have <no uniqueFlag> one child <requiredChild>
  at position <no position>
  custom message: <no customMessage>
```

Parameters:

- requiredChild (Selector[1..n]): specifies the set of selectors that must be matched by the required child element.
- uniqueFlag (UniqueFlag): enumeration mapping strings ‘exactly’ -> true and ‘at least’ -> false. If set to ‘exactly’, the rule will be wiolated if more than one elements match the {requiredChild} selectors.
- Position (integer): specifies the position of the child. If set to ‘0’ the can appear at any index of the investigated element. If set to ’-1’ the specified child must be also the last one.

Default violation message:

- RequiredChildrenError (check id: 46): elements which satisfy .carousel must have exactly one child matching {applyTo}. Elements missing required children: [list of invalid elements]

2.8 Siblings check

This check allows for validating the siblings of the selected elements in various ways.

Alias: siblings check

Syntax:

```
siblings check with id <no id> raises <no type>
  elements <applyTo>
  can <no condition> <allowedSiblings>
  custom message: <no customMessage>
```

Parameters:

- allowedSiblings (Selector[0..n]): applies requirements for the siblings under consideration
- condition (SiblingsCheckType): an enumeration allowing for the following string values:
  -‘not have next siblings’
  -‘not have previous siblings’
  -‘only appear after elements’
  -‘only appear before elements’
  -‘only have next immediate sibling’
  -‘only have previous immediate siblings’
  -‘only appear with at least one sibling’

During the code generation process these options are mapped to appropriate actions. If one of the ‘not have’ options are the allowedSiblings collection must be empty, otherwise it must contain at least one selector. The generated message is different for every condition.

3. SelectorDefinition

SelectorDefinition objects can be children of the root element of LinterDSL. They can hold a collection of Selector objects which can then passed by reference to a Check or another definition with use of a ReferenceSelector. All names of selector definitions must be unique.

Alias: selector definition

Syntax:

```
selector <no name> filters elements <selectors>
```

4. Selector

Selector objects represent CSS selectors without the need of the CSS syntax, using a more descriptive one instead. Selector is an abstract concepts with many subtypes that can be used in the model. Every collection of Selector objects mentioned so far uses ‘or’ as a separator, the only exception will be the GroupSelector.

4.1 AttributeSelector

Translates to standard CSS attribute selector ([attributeName=value]).

Alias: attribute

Syntax:

```
with attribute <no attributeName> = <no value>
```

Parameters:

- attributeName (string): name of the HTML attribute.
- value (string): value of the HTML attribute. If set to '*' the value will not be specified. To look for value = '*' use '/*'.

4.2 ClassSelector

Translates to standard CSS class selector (.className).

Alias: class

Syntax:

```
with class <no value>
```

Parameters:

- value (string): the HTML class.

4.3 TypeSelector

Translates to standard CSS tag selector (tagName).

Alias: type

Syntax:

```
of type <no value>
```

Parameters:

- value (string): the HTML tag.

4.4 NotSelector

Translates to standard CSS not selector (:not(selector)).

Alias: not

Syntax:

```
not <selector>
```

Parameters:

- selector (Selector): other selector to be inverted.

4.5 GroupSelector

Concatinates the generated CSS selectors.

Alias: group

Syntax:

```
not <selector>
```

Parameters:

- selectors (Selector[1..n]): selectors to be concatinated.

4.6 CustomSelector

Allows to input a custom CSS selector with the standard CSS syntax.

Alias: custom

Syntax:

```
matching <no string>
```

Parameters:

- string (string): the CSS selector.

4.7 ClassPatternSelector

Allows to declare a selector filtering elements which match a regular expression.

Alias: pattern

Syntax:

```
with class matching pattern <no regularExpression>
```

Parameters:

- regularExpression (string): elements must have a class which matches this parameter to satisfy this selector.

4.8 ReferenceSelector

Body of a reference selector will be replaced with the body of the referenced selector, allowing for reusing the same definitions.

Alias: reference

Syntax:

```
matching <no selector>
```

Parameters:

- selector (reference to SelectorDefinition): selector definition passed by putting its name.

Appendix C

User guide.

1. Using LinterDSL

LinterDSL is a domain specific language which allows you to describe additional validation rules for frontend CSS frameworks.

In order to use LinterDSL you must first install the Meta Programming System (MPS) from JetBrains. Downloads available at:

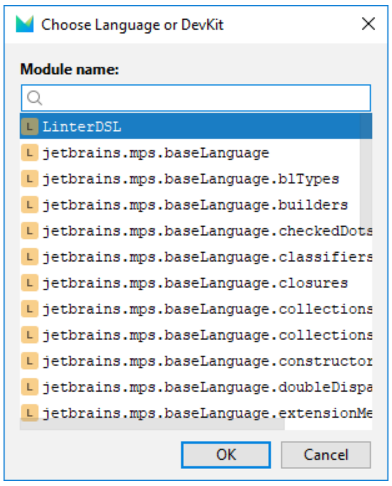
https://www.jetbrains.com/mps/download/

Then download LinterDSL from the following repository:

https://github.com/piskorzm/LinterDSL

Now you can open the downloaded repository as an MPS project. The project contains the LinterDSL language description and an example solution describing rules for the commonly used framework called Bootstrap.

To start defining your own linter, create a new solution with a model. MPS will ask you about the languages used by the model. This is where you must choose LinterDSL.



```
LinterDSL
jetbrains.mps.baseLanguage
jetbrains.mps.baseLanguage.blTypes
jetbrains.mps.baseLanguage.builders
jetbrains.mps.baseLanguage.checkedDots
jetbrains.mps.baseLanguage.classifiers
jetbrains.mps.baseLanguage.closures
jetbrains.mps.baseLanguage.collections
jetbrains.mps.baseLanguage.collections
jetbrains.mps.baseLanguage.constructor
jetbrains.mps.baseLanguage.doubleDispa
jetbrains.mps.baseLanguage.extensionMe
```

Right-click on your new model and choose ‘New->Linter’. You should now see the editor with an empty Linter object. To add new objects use shortcut ‘Ctrl + Space’ and choose available options from the dropdown list or write their alias. For detailed information about the available types of objects in the language, please refer to the documentation.

Once the linter definition is finished, you can generate a script based on it by right-clicking on the solution and choosing the ‘Make Solution’ option. The generated JavaScript file named after the name given to the root element will be placed in:

{ProjectDirectory}/solutions/{SolutionName}/classes_gen/{ModelName}

2. Using generated linters

To use a linter generated by MPS you must create a bookmarklet, which is a bookmark on your browser containing a piece of JavaScript instead of a standard URL.

The command to paste into the URL placeholder is as follows:

```
javascript:(function(){var
s=document.createElement("script");s.onload=function(){linterName(singleIteration
Mode,disabledArray);};s.src="pathToTheLinterFile";document.body.appendChild(s)
})();
```

where:

- singleIterationMode: can be set to either true or false. If set to true every violation will be printed on its own. Otherwise multiple violations of the same rule will be printed as one Error/Warining.
- disabledArray: is an array of check IDs that you want to be omitted. Format: [id1, id2, ...]

To run the validation, open the website that you want to validate and click on the bookmarklet. To see the results open the browser console (press F12).

Appendix D

Bootlint rules mapping to equivalent rules in the version defined with LinterDSL.

#	Bootlint rule ID	Equivalent ID	Deprecated	Outside scope	Missing
1	W001	1	W002	W005	W014
2	W003	2	W004	W013	E015
3	W006	3	W010	E001	
4	W007	4	W011	E007	
5	W008	5	W015	E022	
6	W009	6	E004	E029	
7	W012	7	E008	E036	
8	W016	8	E030	E038	
9	W017	9	E031	E040	
10	W018	10			
11	E002	11			
12	E003	12			
13	E005	13			
14	E006	14			
15	E009	15			
16	E010	16			
17	E011	17			
18	E012	18			
19	E013	19			
20	E014	20			
21	E016	21,22			
22	E017	23,24			
23	E018	25,26			
24	E019	27,28			

25	E020	29			
26	E021	30			
27	E023	31			
28	E024	32			
29	E025	32			
30	E026	33			
31	E027	34			
32	E028	35			
33	E032	36,37,38,39			
34	E033	40			
35	E034	41			
36	E035	42			
37	E037	43			
38	E039	44			
39	E041	45,46			
40	E042	47			
41	E043	48			
42	E044	49,50			
43	E045	51			
44	E046	52			
45	E047	53			
46	E048	52			
47	E049	54			
48	E050	55			
49	E051	56			
50	E052	57			

Appendix E

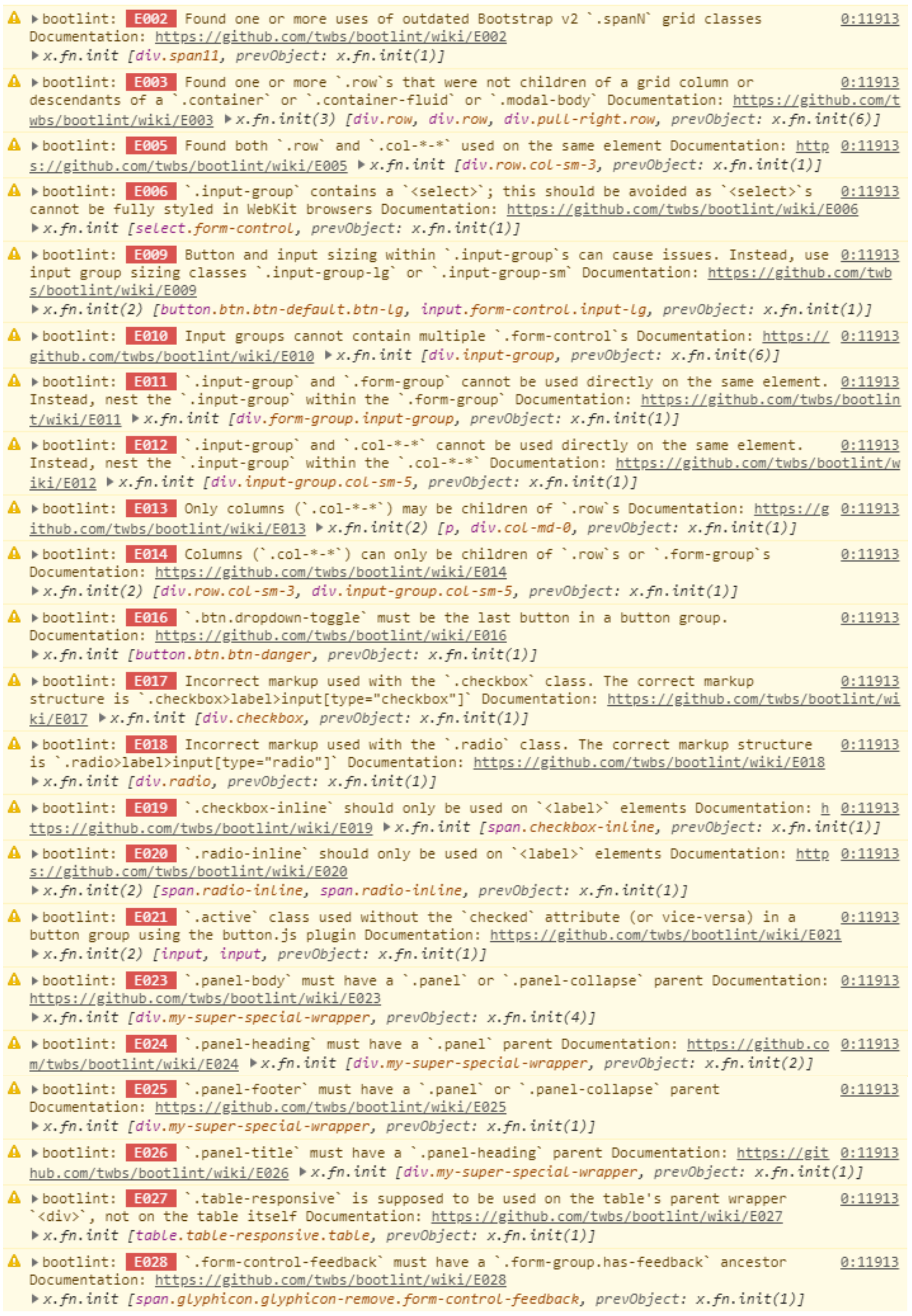
Snippet of Bootlint redefinition in LinterDSL.

```
linter GeneratedBootlint for Bootstrap version 4

reusable selector definitions:

  selector columns filters elements with class matching pattern col-((xs|sm|md|lg)-([1-9]|10|11|12)|([1-9]|10|11|12))
  selector disabledTooltipsAndPopovers filters elements
      matching [disabled][data-toggle="tooltip"] or matching .disabled[data-toggle="tooltip"] or
      matching [disabled][data-toggle="popover"] or matching .disabled[data-toggle="popover"]
  selector nonBodyContainers filters elements matching
      .btn-group [data-toggle="tooltip"]:not([data-container="body"]) or
      matching  .btn-group [data-toggle="popover"]:not([data-container="body"])
  selector oldColumns filters elements with class matching pattern span([1-9]|10|11|12)

validation checks:

  missing tag check with id 1 raises Warning
    framework requires an element ( of type meta and with attribute charset = utf-8 ) or
        ( of type meta and with attribute charset = UTF-8 )
    custom message: -

  missing tag check with id 2 raises Warning
    framework requires an element ( of type meta and with attribute name = viewport and with attribute content = * )
    custom message: -

  invalid element check with id 3 raises Warning
    elements matching disabledTooltipsAndPopovers
    are not valid in the framework
    custom message: -

  element misuse check with id 4 raises Warning
    elements of type button
    must also be elements with attribute type = submit or with attribute type = reset or with attribute type = button
    custom message:  -

  invalid element check with id 5 raises Warning
    elements matching nonBodyContainers
    are not valid in the framework
    custom message: -

  required child check with id 6 raises Warning
    elements matching columns
    must have at least one child of type *
    at position 0
    custom message: -

  required child check with id 7 raises Warning
    elements with class navbar
    must have at least one child with class container or with class container-fluid
    at position 1
    custom message: -

  invalid element check with id 8 raises Warning
    elements ( of type button and with class btn and with attribute disabled = * ) or
        ( of type input and with class btn and with attribute disabled = * )
    are not valid in the framework
    custom message: -

  element misuse check with id 9 raises Warning
    elements of type input
    must also be elements with attribute type = *
    custom message:  -

  element misuse check with id 10 raises Warning
    elements with class matching pattern col-(xs|sm|md|lg)-12
    must also be elements with class matching pattern col-(md|lg)-([1-9](?!.)|10|11)
    custom message:  -

  invalid element check with id 11 raises Error
    elements matching oldColumns
    are not valid in the framework
    custom message: -

  predecessors check with id 12 raises Error
    elements with class row
    must have an ancestor with class container or with class continer-fluid
    0 generations up
    custom message: -
```

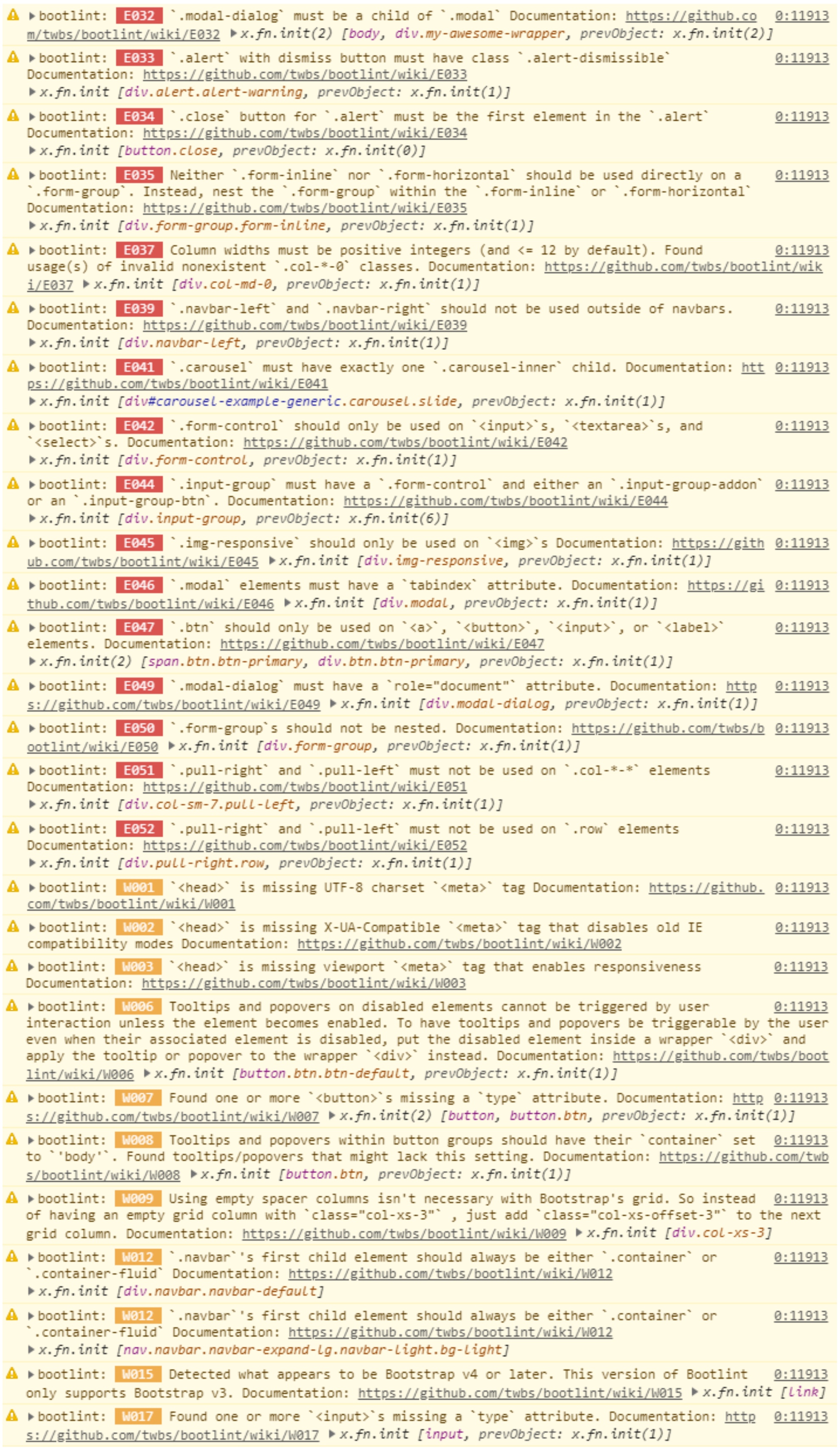
Appendix F

Console log from running original and generated Bootlint on the test page.

Original Bootlint:

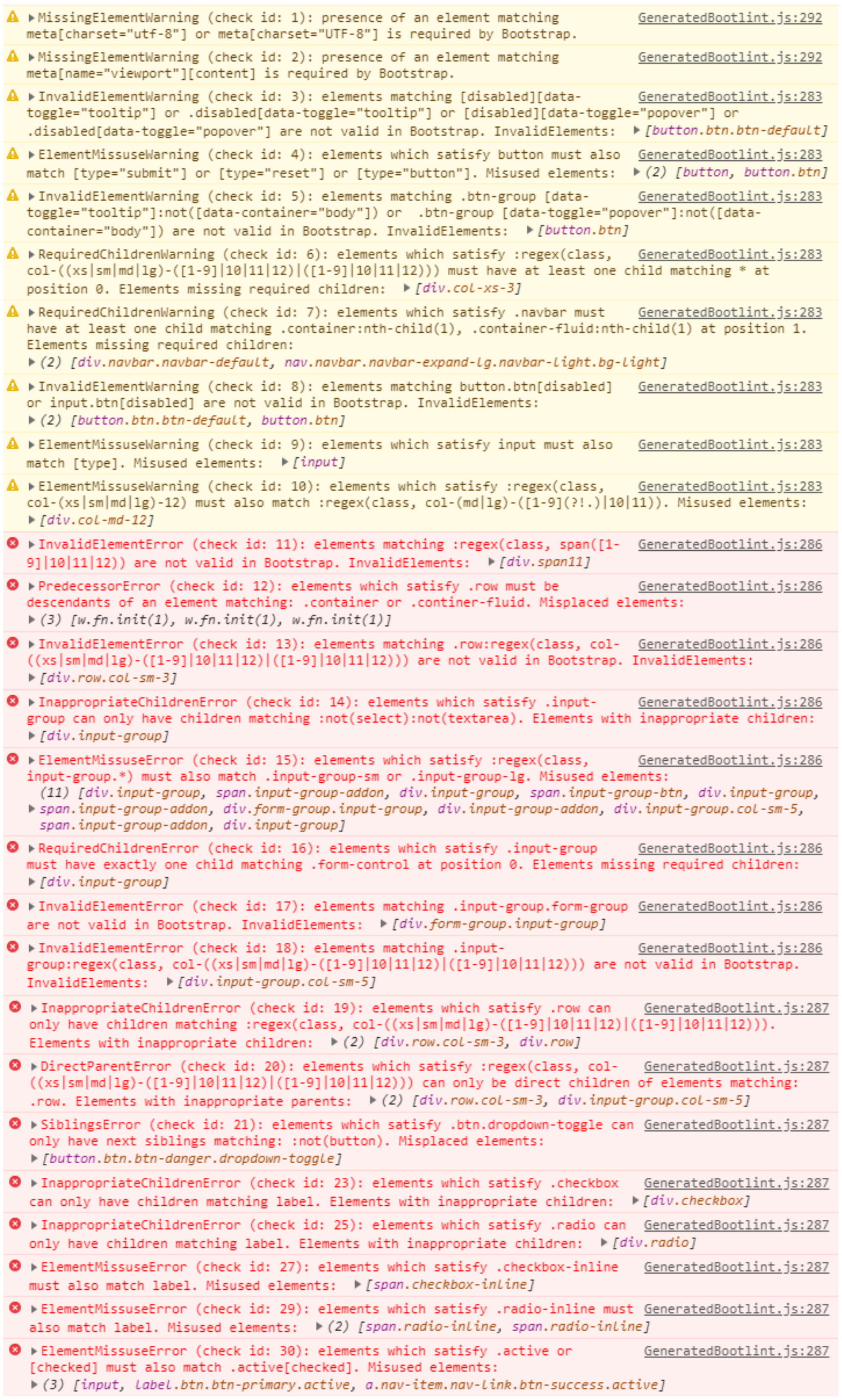


```
⚠ ▶bootlint: E002 Found one or more uses of outdated Bootstrap v2 `.spanN` grid classes        0:11913
  Documentation: https://github.com/twbs/bootlint/wiki/E002
  ▶x.fn.init [div.span11, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E003 Found one or more `.row`s that were not children of a grid column or         0:11913
  descendants of a `.container` or `.container-fluid` or `.modal-body` Documentation: https://github.com/t
  wbs/bootlint/wiki/E003 ▶x.fn.init(3) [div.row, div.row, div.pull-right.row, prevObject: x.fn.init(6)]
⚠ ▶bootlint: E005 Found both `.row` and `.col-*-*` used on the same element Documentation: http 0:11913
  s://github.com/twbs/bootlint/wiki/E005 ▶x.fn.init [div.row.col-sm-3, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E006 `.input-group` contains a `<select>`; this should be avoided as `<select>`s    0:11913
  cannot be fully styled in WebKit browsers Documentation: https://github.com/twbs/bootlint/wiki/E006
  ▶x.fn.init [select.form-control, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E009 Button and input sizing within `.input-group`s can cause issues. Instead, use 0:11913
  input group sizing classes `.input-group-lg` or `.input-group-sm` Documentation: https://github.com/twb
  s/bootlint/wiki/E009
  ▶x.fn.init(2) [button.btn.btn-default.btn-lg, input.form-control.input-lg, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E010 Input groups cannot contain multiple `.form-control`s Documentation: https:// 0:11913
  github.com/twbs/bootlint/wiki/E010 ▶x.fn.init [div.input-group, prevObject: x.fn.init(6)]
⚠ ▶bootlint: E011 `.input-group` and `.form-group` cannot be used directly on the same element. 0:11913
  Instead, nest the `.input-group` within the `.form-group` Documentation: https://github.com/twbs/bootlin
  t/wiki/E011 ▶x.fn.init [div.form-group.input-group, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E012 `.input-group` and `.col-*-*` cannot be used directly on the same element.    0:11913
  Instead, nest the `.input-group` within the `.col-*-*` Documentation: https://github.com/twbs/bootlint/w
  iki/E012 ▶x.fn.init [div.input-group.col-sm-5, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E013 Only columns (`.col-*-*`) may be children of `.row`s Documentation: https://g 0:11913
  ithub.com/twbs/bootlint/wiki/E013 ▶x.fn.init(2) [p, div.col-md-0, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E014 Columns (`.col-*-*`) can only be children of `.row`s or `.form-group`s         0:11913
  Documentation: https://github.com/twbs/bootlint/wiki/E014
  ▶x.fn.init(2) [div.row.col-sm-3, div.input-group.col-sm-5, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E016 `.btn.dropdown-toggle` must be the last button in a button group.             0:11913
  Documentation: https://github.com/twbs/bootlint/wiki/E016
  ▶x.fn.init [button.btn.btn-danger, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E017 Incorrect markup used with the `.checkbox` class. The correct markup          0:11913
  structure is `.checkbox>label>input[type="checkbox"]` Documentation: https://github.com/twbs/bootlint/wi
  ki/E017 ▶x.fn.init [div.checkbox, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E018 Incorrect markup used with the `.radio` class. The correct markup structure   0:11913
  is `.radio>label>input[type="radio"]` Documentation: https://github.com/twbs/bootlint/wiki/E018
  ▶x.fn.init [div.radio, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E019 `.checkbox-inline` should only be used on `<label>` elements Documentation: h 0:11913
  ttps://github.com/twbs/bootlint/wiki/E019 ▶x.fn.init [span.checkbox-inline, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E020 `.radio-inline` should only be used on `<label>` elements Documentation: http 0:11913
  s://github.com/twbs/bootlint/wiki/E020
  ▶x.fn.init(2) [span.radio-inline, span.radio-inline, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E021 `.active` class used without the `checked` attribute (or vice-versa) in a     0:11913
  button group using the button.js plugin Documentation: https://github.com/twbs/bootlint/wiki/E021
  ▶x.fn.init(2) [input, input, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E023 `.panel-body` must have a `.panel` or `.panel-collapse` parent Documentation: 0:11913
  https://github.com/twbs/bootlint/wiki/E023
  ▶x.fn.init [div.my-super-special-wrapper, prevObject: x.fn.init(4)]
⚠ ▶bootlint: E024 `.panel-heading` must have a `.panel` parent Documentation: https://github.co 0:11913
  m/twbs/bootlint/wiki/E024 ▶x.fn.init [div.my-super-special-wrapper, prevObject: x.fn.init(2)]
⚠ ▶bootlint: E025 `.panel-footer` must have a `.panel` or `.panel-collapse` parent              0:11913
  Documentation: https://github.com/twbs/bootlint/wiki/E025
  ▶x.fn.init [div.my-super-special-wrapper, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E026 `.panel-title` must have a `.panel-heading` parent Documentation: https://git 0:11913
  hub.com/twbs/bootlint/wiki/E026 ▶x.fn.init [div.my-super-special-wrapper, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E027 `.table-responsive` is supposed to be used on the table's parent wrapper      0:11913
  `<div>`, not on the table itself Documentation: https://github.com/twbs/bootlint/wiki/E027
  ▶x.fn.init [table.table-responsive.table, prevObject: x.fn.init(1)]
⚠ ▶bootlint: E028 `.form-control-feedback` must have a `.form-group.has-feedback` ancestor      0:11913
  Documentation: https://github.com/twbs/bootlint/wiki/E028
  ▶x.fn.init [span.glyphicon.glyphicon-remove.form-control-feedback, prevObject: x.fn.init(1)]
```

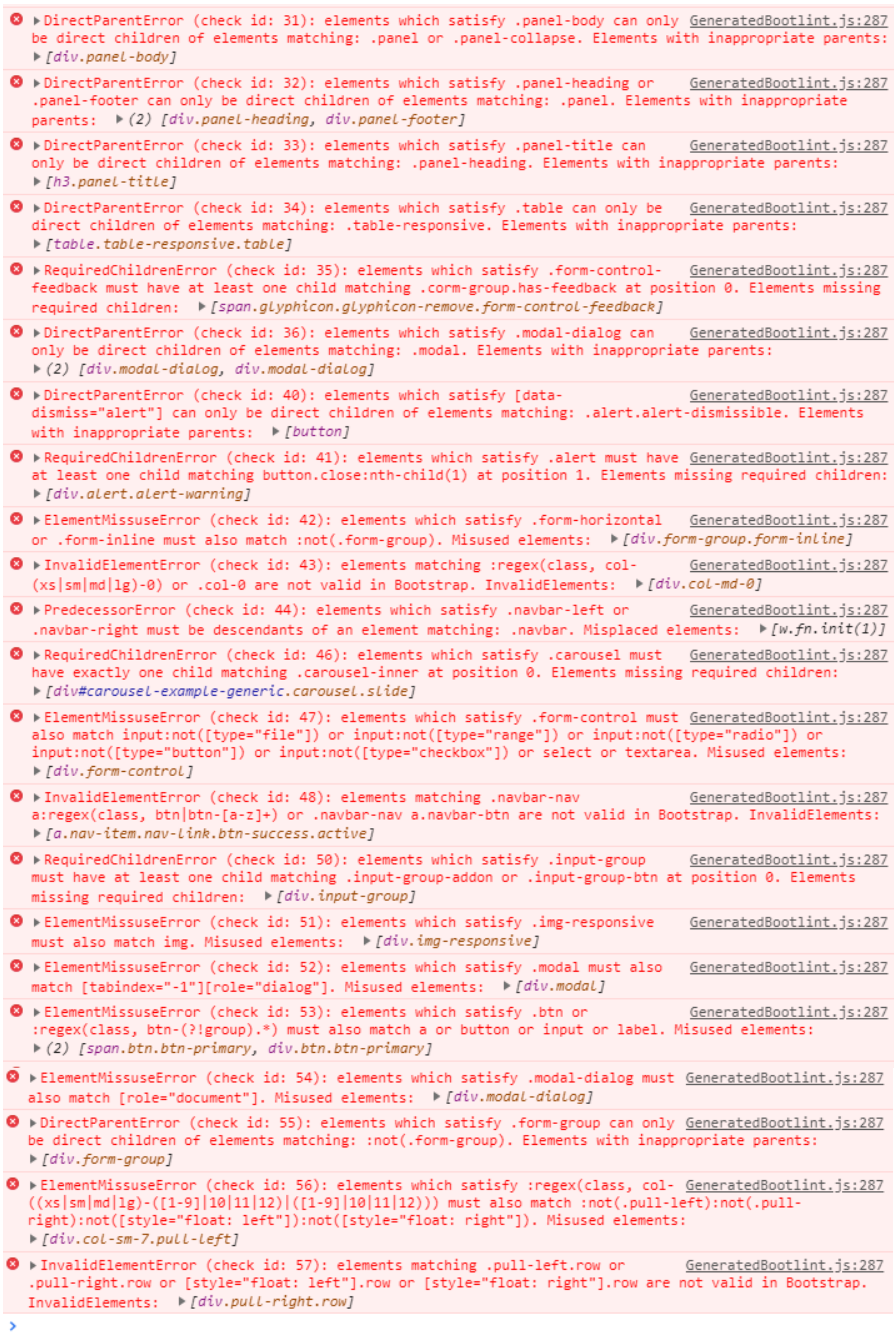


```
▶bootlint: E032 `.modal-dialog` must be a child of `.modal` Documentation: https://github.co   0:11913
m/twbs/bootlint/wiki/E032 ▶x.fn.init(2) [body, div.my-awesome-wrapper, prevObject: x.fn.init(2)]
▶bootlint: E033 `.alert` with dismiss button must have class `.alert-dismissible`              0:11913
Documentation: https://github.com/twbs/bootlint/wiki/E033
▶x.fn.init [div.alert.alert-warning, prevObject: x.fn.init(1)]
▶bootlint: E034 `.close` button for `.alert` must be the first element in the `.alert`          0:11913
Documentation: https://github.com/twbs/bootlint/wiki/E034
▶x.fn.init [button.close, prevObject: x.fn.init(0)]
▶bootlint: E035 Neither `.form-inline` nor `.form-horizontal` should be used directly on a      0:11913
`.form-group`. Instead, nest the `.form-group` within the `.form-inline` or `.form-horizontal`
Documentation: https://github.com/twbs/bootlint/wiki/E035
▶x.fn.init [div.form-group.form-inline, prevObject: x.fn.init(1)]
▶bootlint: E037 Column widths must be positive integers (and <= 12 by default). Found           0:11913
usage(s) of invalid nonexistent `.col-*-0` classes. Documentation: https://github.com/twbs/bootlint/wik
i/E037 ▶x.fn.init [div.col-md-0, prevObject: x.fn.init(1)]
▶bootlint: E039 `.navbar-left` and `.navbar-right` should not be used outside of navbars.       0:11913
Documentation: https://github.com/twbs/bootlint/wiki/E039
▶x.fn.init [div.navbar-left, prevObject: x.fn.init(1)]
▶bootlint: E041 `.carousel` must have exactly one `.carousel-inner` child. Documentation: htt   0:11913
ps://github.com/twbs/bootlint/wiki/E041
▶x.fn.init [div#carousel-example-generic.carousel.slide, prevObject: x.fn.init(1)]
▶bootlint: E042 `.form-control` should only be used on `<input>`s, `<textarea>`s, and           0:11913
`<select>`s. Documentation: https://github.com/twbs/bootlint/wiki/E042
▶x.fn.init [div.form-control, prevObject: x.fn.init(1)]
▶bootlint: E044 `.input-group` must have a `.form-control` and either an `.input-group-addon`   0:11913
or an `.input-group-btn`. Documentation: https://github.com/twbs/bootlint/wiki/E044
▶x.fn.init [div.input-group, prevObject: x.fn.init(6)]
▶bootlint: E045 `.img-responsive` should only be used on `<img>`s Documentation: https://gith   0:11913
ub.com/twbs/bootlint/wiki/E045 ▶x.fn.init [div.img-responsive, prevObject: x.fn.init(1)]
▶bootlint: E046 `.modal` elements must have a `tabindex` attribute. Documentation: https://gi   0:11913
thub.com/twbs/bootlint/wiki/E046 ▶x.fn.init [div.modal, prevObject: x.fn.init(1)]
▶bootlint: E047 `.btn` should only be used on `<a>`, `<button>`, `<input>`, or `<label>`        0:11913
elements. Documentation: https://github.com/twbs/bootlint/wiki/E047
▶x.fn.init(2) [span.btn.btn-primary, div.btn.btn-primary, prevObject: x.fn.init(1)]
▶bootlint: E049 `.modal-dialog` must have a `role="document"` attribute. Documentation: http    0:11913
s://github.com/twbs/bootlint/wiki/E049 ▶x.fn.init [div.modal-dialog, prevObject: x.fn.init(1)]
▶bootlint: E050 `.form-group`s should not be nested. Documentation: https://github.com/twbs/b   0:11913
ootlint/wiki/E050 ▶x.fn.init [div.form-group, prevObject: x.fn.init(1)]
▶bootlint: E051 `.pull-right` and `.pull-left` must not be used on `.col-*-*` elements          0:11913
Documentation: https://github.com/twbs/bootlint/wiki/E051
▶x.fn.init [div.col-sm-7.pull-left, prevObject: x.fn.init(1)]
▶bootlint: E052 `.pull-right` and `.pull-left` must not be used on `.row` elements              0:11913
Documentation: https://github.com/twbs/bootlint/wiki/E052
▶x.fn.init [div.pull-right.row, prevObject: x.fn.init(1)]
▶bootlint: W001 `<head>` is missing UTF-8 charset `<meta>` tag Documentation: https://github.   0:11913
com/twbs/bootlint/wiki/W001
▶bootlint: W002 `<head>` is missing X-UA-Compatible `<meta>` tag that disables old IE           0:11913
compatibility modes Documentation: https://github.com/twbs/bootlint/wiki/W002
▶bootlint: W003 `<head>` is missing viewport `<meta>` tag that enables responsiveness           0:11913
Documentation: https://github.com/twbs/bootlint/wiki/W003
▶bootlint: W006 Tooltips and popovers on disabled elements cannot be triggered by user          0:11913
interaction unless the element becomes enabled. To have tooltips and popovers be triggerable by the user
even when their associated element is disabled, put the disabled element inside a wrapper `<div>` and
apply the tooltip or popover to the wrapper `<div>` instead. Documentation: https://github.com/twbs/boot
lint/wiki/W006 ▶x.fn.init [button.btn.btn-default, prevObject: x.fn.init(1)]
▶bootlint: W007 Found one or more `<button>`s missing a `type` attribute. Documentation: http   0:11913
s://github.com/twbs/bootlint/wiki/W007 ▶x.fn.init(2) [button, button.btn, prevObject: x.fn.init(1)]
▶bootlint: W008 Tooltips and popovers within button groups should have their `container` set    0:11913
to `'body'`. Found tooltips/popovers that might lack this setting. Documentation: https://github.com/twb
s/bootlint/wiki/W008 ▶x.fn.init [button.btn, prevObject: x.fn.init(1)]
▶bootlint: W009 Using empty spacer columns isn't necessary with Bootstrap's grid. So instead     0:11913
of having an empty grid column with `class="col-xs-3"` , just add `class="col-xs-offset-3"` to the next
grid column. Documentation: https://github.com/twbs/bootlint/wiki/W009 ▶x.fn.init [div.col-xs-3]
▶bootlint: W012 `.navbar`'s first child element should always be either `.container` or        0:11913
`.container-fluid` Documentation: https://github.com/twbs/bootlint/wiki/W012
▶x.fn.init [div.navbar.navbar-default]
▶bootlint: W012 `.navbar`'s first child element should always be either `.container` or        0:11913
`.container-fluid` Documentation: https://github.com/twbs/bootlint/wiki/W012
▶x.fn.init [nav.navbar.navbar-expand-lg.navbar-light.bg-light]
▶bootlint: W015 Detected what appears to be Bootstrap v4 or later. This version of Bootlint     0:11913
only supports Bootstrap v3. Documentation: https://github.com/twbs/bootlint/wiki/W015 ▶x.fn.init [link]
▶bootlint: W017 Found one or more `<input>`s missing a `type` attribute. Documentation: http    0:11913
s://github.com/twbs/bootlint/wiki/W017 ▶x.fn.init [input, prevObject: x.fn.init(1)]
```

Generated Bootlint:



```
▸MissingElementWarning (check id: 1): presence of an element matching          GeneratedBootlint.js:292
meta[charset="utf-8"] or meta[charset="UTF-8"] is required by Bootstrap.
▸MissingElementWarning (check id: 2): presence of an element matching          GeneratedBootlint.js:292
meta[name="viewport"][content] is required by Bootstrap.
▸InvalidElementWarning (check id: 3): elements matching [disabled][data-       GeneratedBootlint.js:283
toggle="tooltip"] or .disabled[data-toggle="tooltip"] or [disabled][data-toggle="popover"] or
.disabled[data-toggle="popover"] are not valid in Bootstrap. InvalidElements:  ▸[button.btn.btn-default]
▸ElementMissuseWarning (check id: 4): elements which satisfy button must also  GeneratedBootlint.js:283
match [type="submit"] or [type="reset"] or [type="button"]. Misused elements:  ▸(2) [button, button.btn]
▸InvalidElementWarning (check id: 5): elements matching .btn-group [data-      GeneratedBootlint.js:283
toggle="tooltip"]:not([data-container="body"]) or  .btn-group [data-toggle="popover"]:not([data-
container="body"]) are not valid in Bootstrap. InvalidElements:  ▸[button.btn]
▸RequiredChildrenWarning (check id: 6): elements which satisfy :regex(class,   GeneratedBootlint.js:283
col-((xs|sm|md|lg)-([1-9]|10|11|12)|([1-9]|10|11|12))) must have at least one child matching * at
position 0. Elements missing required children:  ▸[div.col-xs-3]
▸RequiredChildrenWarning (check id: 7): elements which satisfy .navbar must    GeneratedBootlint.js:283
have at least one child matching .container:nth-child(1), .container-fluid:nth-child(1) at position 1.
Elements missing required children:
▸(2) [div.navbar.navbar-default, nav.navbar.navbar-expand-lg.navbar-light.bg-light]
▸InvalidElementWarning (check id: 8): elements matching button.btn[disabled]   GeneratedBootlint.js:283
or input.btn[disabled] are not valid in Bootstrap. InvalidElements:
▸(2) [button.btn.btn-default, button.btn]
▸ElementMissuseWarning (check id: 9): elements which satisfy input must also   GeneratedBootlint.js:283
match [type]. Misused elements:  ▸[input]
▸ElementMissuseWarning (check id: 10): elements which satisfy :regex(class,    GeneratedBootlint.js:283
col-(xs|sm|md|lg)-12) must also match :regex(class, col-(md|lg)-([1-9](?!.)|10|11)). Misused elements:
▸[div.col-md-12]
▸InvalidElementError (check id: 11): elements matching :regex(class, span([1-  GeneratedBootlint.js:286
9]|10|11|12)) are not valid in Bootstrap. InvalidElements:  ▸[div.span11]
▸PredecessorError (check id: 12): elements which satisfy .row must be          GeneratedBootlint.js:286
descendants of an element matching: .container or .continer-fluid. Misplaced elements:
▸(3) [w.fn.init(1), w.fn.init(1), w.fn.init(1)]
▸InvalidElementError (check id: 13): elements matching .row:regex(class, col-  GeneratedBootlint.js:286
((xs|sm|md|lg)-([1-9]|10|11|12)|([1-9]|10|11|12))) are not valid in Bootstrap. InvalidElements:
▸[div.row.col-sm-3]
▸InappropriateChildrenError (check id: 14): elements which satisfy .input-     GeneratedBootlint.js:286
group can only have children matching :not(select):not(textarea). Elements with inappropriate children:
▸[div.input-group]
▸ElementMissuseError (check id: 15): elements which satisfy :regex(class,      GeneratedBootlint.js:286
input-group.*) must also match .input-group-sm or .input-group-lg. Misused elements:
 (11) [div.input-group, span.input-group-addon, div.input-group, span.input-group-btn, div.input-group,
▸span.input-group-addon, div.form-group.input-group, div.input-group-addon, div.input-group.col-sm-5,
 span.input-group-addon, div.input-group]
▸RequiredChildrenError (check id: 16): elements which satisfy .input-group     GeneratedBootlint.js:286
must have exactly one child matching .form-control at position 0. Elements missing required children:
▸[div.input-group]
▸InvalidElementError (check id: 17): elements matching .input-group.form-group GeneratedBootlint.js:286
are not valid in Bootstrap. InvalidElements:  ▸[div.form-group.input-group]
▸InvalidElementError (check id: 18): elements matching .input-                 GeneratedBootlint.js:286
group:regex(class, col-((xs|sm|md|lg)-([1-9]|10|11|12)|([1-9]|10|11|12))) are not valid in Bootstrap.
InvalidElements:  ▸[div.input-group.col-sm-5]
▸InappropriateChildrenError (check id: 19): elements which satisfy .row can    GeneratedBootlint.js:287
only have children matching :regex(class, col-((xs|sm|md|lg)-([1-9]|10|11|12)|([1-9]|10|11|12))).
Elements with inappropriate children:  ▸(2) [div.row.col-sm-3, div.row]
▸DirectParentError (check id: 20): elements which satisfy :regex(class, col-   GeneratedBootlint.js:287
((xs|sm|md|lg)-([1-9]|10|11|12)|([1-9]|10|11|12))) can only be direct children of elements matching:
.row. Elements with inappropriate parents:  ▸(2) [div.row.col-sm-3, div.input-group.col-sm-5]
▸SiblingsError (check id: 21): elements which satisfy .btn.dropdown-toggle can GeneratedBootlint.js:287
only have next siblings matching: :not(button). Misplaced elements:
▸[button.btn.btn-danger.dropdown-toggle]
▸InappropriateChildrenError (check id: 23): elements which satisfy .checkbox   GeneratedBootlint.js:287
can only have children matching label. Elements with inappropriate children:  ▸[div.checkbox]
▸InappropriateChildrenError (check id: 25): elements which satisfy .radio can  GeneratedBootlint.js:287
only have children matching label. Elements with inappropriate children:  ▸[div.radio]
▸ElementMissuseError (check id: 27): elements which satisfy .checkbox-inline   GeneratedBootlint.js:287
must also match label. Misused elements:  ▸[span.checkbox-inline]
▸ElementMissuseError (check id: 29): elements which satisfy .radio-inline must GeneratedBootlint.js:287
also match label. Misused elements:  ▸(2) [span.radio-inline, span.radio-inline]
▸ElementMissuseError (check id: 30): elements which satisfy .active or         GeneratedBootlint.js:287
[checked] must also match .active[checked]. Misused elements:
▸(3) [input, label.btn.btn-primary.active, a.nav-item.nav-link.btn-success.active]
```



```
▸DirectParentError (check id: 31): elements which satisfy .panel-body can only  GeneratedBootlint.js:287
be direct children of elements matching: .panel or .panel-collapse. Elements with inappropriate parents:
▸[div.panel-body]
▸DirectParentError (check id: 32): elements which satisfy .panel-heading or      GeneratedBootlint.js:287
.panel-footer can only be direct children of elements matching: .panel. Elements with inappropriate
parents:  ▸(2) [div.panel-heading, div.panel-footer]
▸DirectParentError (check id: 33): elements which satisfy .panel-title can       GeneratedBootlint.js:287
only be direct children of elements matching: .panel-heading. Elements with inappropriate parents:
▸[h3.panel-title]
▸DirectParentError (check id: 34): elements which satisfy .table can only be     GeneratedBootlint.js:287
direct children of elements matching: .table-responsive. Elements with inappropriate parents:
▸[table.table-responsive.table]
▸RequiredChildrenError (check id: 35): elements which satisfy .form-control-     GeneratedBootlint.js:287
feedback must have at least one child matching .corm-group.has-feedback at position 0. Elements missing
required children:  ▸[span.glyphicon.glyphicon-remove.form-control-feedback]
▸DirectParentError (check id: 36): elements which satisfy .modal-dialog can      GeneratedBootlint.js:287
only be direct children of elements matching: .modal. Elements with inappropriate parents:
▸(2) [div.modal-dialog, div.modal-dialog]
▸DirectParentError (check id: 40): elements which satisfy [data-                 GeneratedBootlint.js:287
dismiss="alert"] can only be direct children of elements matching: .alert.alert-dismissible. Elements
with inappropriate parents:  ▸[button]
▸RequiredChildrenError (check id: 41): elements which satisfy .alert must have  GeneratedBootlint.js:287
at least one child matching button.close:nth-child(1) at position 1. Elements missing required children:
▸[div.alert.alert-warning]
▸ElementMissuseError (check id: 42): elements which satisfy .form-horizontal    GeneratedBootlint.js:287
or .form-inline must also match :not(.form-group). Misused elements:  ▸[div.form-group.form-inline]
▸InvalidElementError (check id: 43): elements matching :regex(class, col-       GeneratedBootlint.js:287
(xs|sm|md|lg)-0) or .col-0 are not valid in Bootstrap. InvalidElements:  ▸[div.col-md-0]
▸PredecessorError (check id: 44): elements which satisfy .navbar-left or         GeneratedBootlint.js:287
.navbar-right must be descendants of an element matching: .navbar. Misplaced elements:  ▸[w.fn.init(1)]
▸RequiredChildrenError (check id: 46): elements which satisfy .carousel must    GeneratedBootlint.js:287
have exactly one child matching .carousel-inner at position 0. Elements missing required children:
▸[div#carousel-example-generic.carousel.slide]
▸ElementMissuseError (check id: 47): elements which satisfy .form-control must GeneratedBootlint.js:287
also match input:not([type="file"]) or input:not([type="range"]) or input:not([type="radio"]) or
input:not([type="button"]) or input:not([type="checkbox"]) or select or textarea. Misused elements:
▸[div.form-control]
▸InvalidElementError (check id: 48): elements matching .navbar-nav               GeneratedBootlint.js:287
a:regex(class, btn|btn-[a-z]+) or .navbar-nav a.navbar-btn are not valid in Bootstrap. InvalidElements:
▸[a.nav-item.nav-link.btn-success.active]
▸RequiredChildrenError (check id: 50): elements which satisfy .input-group      GeneratedBootlint.js:287
must have at least one child matching .input-group-addon or .input-group-btn at position 0. Elements
missing required children:  ▸[div.input-group]
▸ElementMissuseError (check id: 51): elements which satisfy .img-responsive     GeneratedBootlint.js:287
must also match img. Misused elements:  ▸[div.img-responsive]
▸ElementMissuseError (check id: 52): elements which satisfy .modal must also    GeneratedBootlint.js:287
match [tabindex="-1"][role="dialog"]. Misused elements:  ▸[div.modal]
▸ElementMissuseError (check id: 53): elements which satisfy .btn or             GeneratedBootlint.js:287
:regex(class, btn-(?!group).*) must also match a or button or input or label. Misused elements:
▸(2) [span.btn.btn-primary, div.btn.btn-primary]
▸ElementMissuseError (check id: 54): elements which satisfy .modal-dialog must GeneratedBootlint.js:287
also match [role="document"]. Misused elements:  ▸[div.modal-dialog]
▸DirectParentError (check id: 55): elements which satisfy .form-group can only GeneratedBootlint.js:287
be direct children of elements matching: :not(.form-group). Elements with inappropriate parents:
▸[div.form-group]
▸ElementMissuseError (check id: 56): elements which satisfy :regex(class, col- GeneratedBootlint.js:287
((xs|sm|md|lg)-([1-9]|10|11|12)|([1-9]|10|11|12))) must also match :not(.pull-left):not(.pull-
right):not([style="float: left"]):not([style="float: right"]). Misused elements:
▸[div.col-sm-7.pull-left]
▸InvalidElementError (check id: 57): elements matching .pull-left.row or        GeneratedBootlint.js:287
.pull-right.row or [style="float: left"].row or [style="float: right"].row are not valid in Bootstrap.
InvalidElements:  ▸[div.pull-right.row]
>
```

Bibliography

[1] “HTML 5.2”, W3C Recommendation, [Online]. Available: https://www.w3.org/TR/html [Accessed 5-Feb-2019]

[2] D. Goodman, Dynamic HTML the definitive reference, O’Reilly, 1998.

[3] “Learn to style HTML using CSS”, MDN Web Docs. Available: https://developer.mozilla.org/en-US/docs/Learn/CSS [Accessed 5-Feb-2019]

[4] Håkon W Lie, “Cascading HTML style sheets - a proposal”, 1994 [Online]. Available: https://www.w3.org/People/howcome/p/cascade.html

[5] D. Shea and M. Holzschlag, “The Zen of CSS design”, Berkeley, CA: New Riders, 2005.

[6] “Cascading style sheets”, Wikipedia, The Free Encyclopedia, [Online] Available: https://en.wikipedia.org/wiki/Cascading_Style_Sheets [Accessed 5-Feb-2019]

[7] “CSS: Selectors”, W3C Recommendation, [Online]. Available https://www.w3.org/TR/CSS21/selector.html [Accessed 5-Feb-2019]

[8] “CSS: Full property table”, W3C Recommendation, [Online]. Available: https://www.w3.org/TR/CSS2/propidx.html [Accessed 5-Feb-2019]

[9] “What are Frameworks? 22 Best Responsive CSS Frameworks for Web Design”, Awwwards, [Online]. Available: https://www.awwwards.com/what-are-frameworks-22-best-responsive-css-frameworks-for-web-design.html [Accessed: 7-Feb-2019]

[10] “Bootstrap from Twitter”, Developer Blog, [Online]. Available: https://blog.twitter.com/developer/en_us/a/2011/bootstrap-twitter.html [Accessed 8-Feb-2019]

[11] “Bootstrap 4 Grid System”, W3School, [Online]. Available: https://www.w3schools.com/bootstrap4/bootstrap_grid_system.asp [Accessed 10-Feb-2019]

[12] S. C. Johnson, Lint, a C program checker, Bell Telephone Laboratories, 1977.

[13] ESLint software, [Online]. Available: https://eslint.org/

[14] ESLint rule list, https://eslint.org/docs/rules/.

[15] “Bootlint documentation”, GitHub, [Online]. Available: https://github.com/twbs/bootlint/wiki

[16] A. G. Kleppe, Software language engineering, Addison-Wesley, 2009.

[17] jQuery JavaScript library, [Online]. Available: https://jquery.com/

[18] “Regex Selector for jQuery”, James Podolsey, [Online]. Available: https://j11y.io/javascript/regex-selector-for-jquery/ [Accessed 20-Mar-2019]

[19] LinterDSL repository, Marcin Piskorz, GitHub, [Online]. Available: https://github.com/piskorzm/LinterDSL