Detecting Violations of CSS Code Conventions

Boryana Goncharenko
boryana.goncharenko@gmail.com

July 17, 2015, 21 pages

Supervisor: Vadim Zaytsev
Host organisation: University of Amsterdam

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

This section summarises the content of the thesis for potential readers who do not have time to read it whole, or for those undecided whether to read it at all. Sum up the following aspects:

- relevance and motivation for the research
- research question(s) and a brief description of the research method
- results, contributions and conclusions

Chapter 1

Introduction

Code conventions put constraints on how code should be written in the context of a project, organization or programming language. Style guides can comprise conventions that refer to whitespacing, indentation, code layout, preference of syntactic structures, code patterns and anti- patterns. They are mainly used to achieve code consistency, which in turn improves the readability, understandability and maintainability of the code [Citations].

Style guides are often designed in an ad hoc manner. Coding conventions typically live in documents that contain a description of each rule in natural language accompanied by code examples. This is the case with the style guidelines of Mozilla [8], Google [6], GitHub [10], WordPress [11] and Drupal [4]. To apply the conventions, developers first need to read, understand and apply them manually. Such an approach introduces a number of issues. Using natural language can make guidelines incorrect, ambiguous, implicit or too general. Another problem is that developers apply conventions manually, which increases the chances of introducing violations involuntarily. There are tools that check for compliance with guidelines, however, they are often hard to customize or limited to one type of violations, e.g. only whitespacing.

The core idea behind the project is to provide a solution that lets developers express an arbitrary set of coding conventions and detect their violations automatically in an IDE. Writing conventions in an executable form could assist authors in detecting incorrect, ambiguous or inconsistent guidelines. Automatic detection of violations could minimize the effort required by developers to write code that complies to the guidelines. To meet the constraints of a Masters project, the implementation is limited to the domain of Cascading Style Sheets (CSS). The project requires determining the need for CSS code conventions in organizations, collecting and analyzing available style guides, and providing a way to express conventions. Specifically, the project attempts to answer the following set of questions:

- **Research Question 1:** Do developers still maintain plain CSS?
- **Research Question 2:** What code conventions for CSS exist?
- **Research Question 3:** How to express existing CSS code conventions?

The thesis is organized as follows. Chapter 2 contains background notions and terms used throughout the thesis. The method and results for RQ1 are presented in Chapter 3. Discovering the existing code conventions is illustrates in Chapter 4. The design and description of a DSL is presented in Chapter 5. Chapter 6 concludes the thesis.

Chapter 2

Background

2.1 Bunge-Wand-Weber ontology

A **conceptualization** is an abstract, simplified view of the world that is represented for some purpose [Gru95]. It consists of the concepts that are assumed to exist in some area of interest and their relationships [Gru95]. An **ontology** is an explicit specification of a conceptualization [Gru95]. It describes what is fundamental in the totality of what exists and it defines the most general categories to which we need to refer in constructing a description of reality [MS04].

Milton further distinguishes between two kinds of ontologies: top-level and domain-specific [MS04]. Ontologies of the former type are highly general and provide the theoretical foundations for representation and modeling of systems. Ontologies of the latter type are restricted to define concepts and their relations that fall in a particular domain. Every domain-specific ontology needs to use categories that are captured in a top-level ontology. In this sense a domain-specific ontology is based on a specific top-level ontology [MS04].

The Bunge-Wand-Weber (BWW) ontology [WW90] is a high-level ontology used in the representation model developed by Wand and Weber [WW95]. Table 2.1 presents a selected set of the ontological constructs in the BWW ontology.

Table 2.1: Selected ontological constructs in the BWW representation model

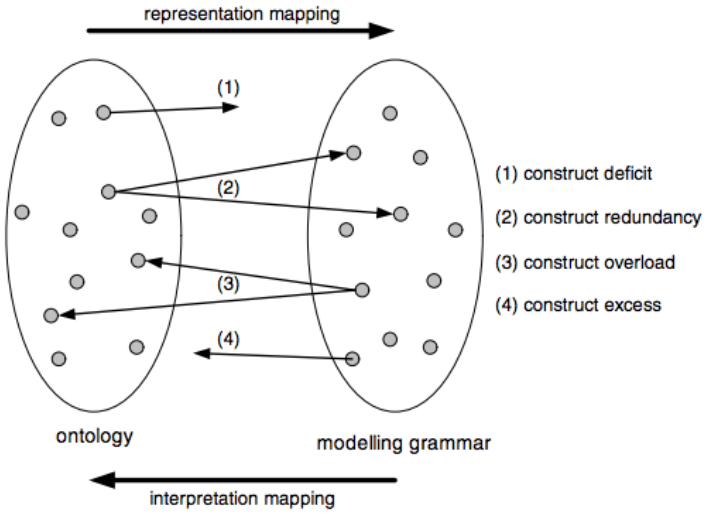
Ontological construct	**Explanation**
Thing	The elementary unit in the BWW ontological model. The real world is made up of things. A composite thing may be made up of other things (composite or primitive)
Properties	Things possess properties. A property is modeled via a function that maps the thing into some value. A property of a composite thing that belongs to a component thing is called an hereditary property. Otherwise it is called an emergent property. A property that is inherently a property of an individual thing is called an intrinsic property. A property that is meaningful only in the context of two or mode things is called a mutual or relational property.
State	A vector of values for all property functions of a thing
Event	A change of state of a thing. It is effected via a transformation.
Transformation	A mapping from a domain comprising states to a codomain comprising states
History	The chronologically ordered states that a thing traverses

Coupling	A thing acts on another thing if its existence affects the history of the other thing. The two things are said to be coupled or interact.
Class	A class is a set of things that can be defined via their possessing a characteristic property.
Kind	A kind is a set of things that can be defined only via their possessing two or more properties.
System	A set of things is a system if, for any bi-partitioning of the set, couplings exist among things in the two subsets.
System Composition	The things in the system are its composition.
System Environment	Things that are not in the system but interact with things in the system are called the environment of the system.

2.2 Ontological analysis

Ontological analysis is an established approach for evaluating the quality of software engineering notations [Moo09]. It consists of a two way comparison between a set of modeling grammar constructs and a set of ontological constructs. The **interpretation mapping** compares the notation with the ontology and the **representation mapping** compares the ontology with the notation [GE07]. The fundamental premise of ontological analysis is that modeling grammars are incomplete if they are not able to represent what exists in reality [GR00]. Furthermore, there should be a one-to-one mapping between the modeling grammar and the ontological constructs. Any deviation from such correspondence leads to an anomaly (Figure 2.1).

Figure 2.1: Ontological Analysis [GE07]



Construct deficit occurs when an ontological construct does not have a corresponding construct in the modeling grammar. **Construct redundancy** is observed when a single ontological construct maps to more than one modeling grammar construct. **Construct overload** appears when a modeling grammar construct corresponds to more than one ontological construct. **Construct excess** occurs when a modeling grammar construct does not map to any ontological construct. [Moo09]

Chapter 3

Evaluating the Need for CSS Code Conventions

I will talk about answering the first research question.

3.1 Research Method

Despite the new features added in the second [1] and third [5] versions of CSS, the language has obvious limita- tions, e.g. lack of variables. A number of preprocessors have evolved to tackle the downsides of CSS. Solutions such as SASS [3], LESS [9] and Stylus [7] offer enhanced or even different syntax and translate it to CSS. Pre- processors are not only ubiquitously recommended, but also widely adopted in practice. Obviously, using such solutions avoids the need for CSS code conventions because the code is generated and not maintained directly. Thus, CSS code conventions make sense only if developers handcraft the CSS files.

To determine whether CSS is written and maintained as opposed to being generated, all commits to open source repositories hosted on GitHub for 2015 (up to April) are being analyzed. If the commit contains a file with extension .scss, .sass, .less or .styl, it is considered preprocessor maintenance. In case the commit contains files with the .css extension and no preprocessor extensions, it is considered maintenance of CSS. To exclude cases in which developers commit third- party CSS, only commits that modify CSS files are taken into consideration. Commits that add or delete CSS files are ignored.

3.2 Results

Despite of the popular belief that nowadays preprocessors are prevailing than CSS, results illustrate plain CSS is still used. Figure 1 summarizes the findings.

Having the above in mind I assume that CSS is still maintained in practice.

3.3 Analysis

What if the main users of preprocessors are private?

What if the time interval was too short?

What if there are other preprocessor that are not counted?

Chapter 4

Discovering Existing CSS Code Conventions

4.1 Research Method

The CSS community has produced a pool of recommendations, best practices, and style guides, but how to choose the among them? Since the primary organization responsible for the specification of CSS has not recommended code conventions, any selection strategy based on the author of the conventions could be considered cherry picking.

To determine the set of code conventions, two searches with the keywords CSS code conventions have been made using the search engines http://duckduckgo.com and http://google.com. The first 50 results of each search have been analyzed. From each result only conventions about pure CSS are taken into account and guidelines for CSS preprocessors are ignored. In case the result is an online magazine or a blog post that links to other resources, these references are considered as results and analyzed separately.

Discuss problem with conventions here:

Overgeneralized conventions - The description of the convention is too general to be applied, e.g. “dont use CSS hacks try a different approach first”.

Incorrect conventions - There is a discrepancy between the description of the rule and the provided example. An instance of such contradiction is when the convention nothing but declarations should be indented is followed by a code snippet illustrating that rules in media queries should also be indented.

Ambiguous conventions - There is more than one interpretation of a convention. For example, rules with more than 4 selectors are not allowed could be seen as forbidding multi- selectors with more than four selectors, or disallowing selectors with more than four simple- selectors.

Implicit conventions - There are rules that are not explicitly stated and could only be inferred by the other rules. For example, the convention you can put values on multiple lines is not preceded by a convention that requires values to appear on one line.

Inconsistencies between conventions in one ruleset - Google say that charsets should not be used in css, but later say that in charsets double quotes should be used. Surely, this opens the question - so when do I need to use charsets?

4.2 Results

Results of the two searches include the CSS coding guidelines of CSS professionals as well as leading companies, e.g. Google, Mozilla, GitHub, Wordpress. The accumulated corpus consists of 165 unique coding conventions. . Here is an example:

A full list is available at this GitHub page.

4.3 Analysis

There are conventions that cannot be detected at all?

What if not everyone have published their style guides? Well, some respectful companies did.

Chapter 5

Expressing CSS Code Conventions

5.1 Analysis of conventions corpus

Code conventions is an umbrella term that comprises a vast set of rules. To check what the CSS conventions are all about, I analyzed the conventions and it turned out there are three broad groups of conventions. In general, CSS conventions refer to whitespacing and indentation, to syntax preference or to programming style.

Further, the analysis consists of analyzing each individual convention.

The approach used to explore the domain consists of analyzing how the current system works. In other words, the analysis aims at revealing how developers detect violations manually, what is knowledge developers need and what are the particular steps they make. The domain analysis phase consists of two steps:

- Determine possible violations. In this step the meaning of the convention is discussed and violations are made explicit.
- List the specific actions that need to be taken to detect violations manually. Developers check their code for compliance manually. To perform such checks, developers need to understand different concepts, e.g. the concept of a rule, html element, ids, etc and perform certain actions, such as find a structure, evaluate a constraint etc.

The following analysis of three conventions illustrates the process described above. Analysis of all conventions in the corpus are available at GitHub.

Convention: Disallow empty rules.

Author: CSS lint

Violations: Presence of rulesets that do not contain declarations. In case at least one declaration is present, the ruleset does not violate the convention. Examples include:

```
1 .myclass { } /* violation */
2 .myclass { /* Comment */ } /* violation */
3 .myclass { color: green; } /* not violation */
```

Actions: Recognize rulesets and declarations. Determine whether a ruleset does not contain any declarations.

This convention aims at getting rid of one type of refactoring leftovers - rulesets without declarations [CSS lint]. Removing empty rulesets reduces the total size of CSS that needs to be processed by the browser. One possible approach for discovering violations of the convention at hand is to search the

stylesheet for rulesets and then check whether each ruleset contains a declaration. To perform this search successfully, developers need to understand the concept of a ruleset and a declaration, i.e. they need to be able to recognize these two CSS structures. Further, developers need to determine relations between structures, particularly, whether a ruleset contains a declaration.

In this way, all conventions have been analyzed and as a result, the following set of conclusions have been made:

1) Every convention imposes a set of constraints over the program. Violations occur when these constraints are not met.

2) Conventions can refer to nodes in abstract syntax tree, concrete syntax tree and parse tree of CSS.

3) Conventions can refer to nodes based on their type or function in the program (hex, color)

4) Conventions can refer to nodes based on CSS-specific knowledge, e.g. is vendor-specific property

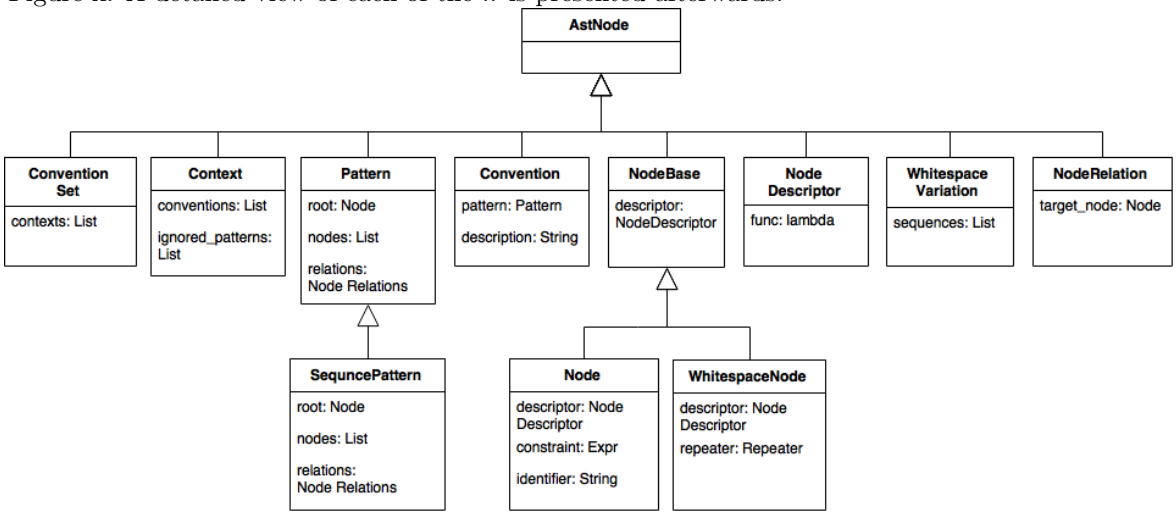
4) Conventions can refer to nodes based on their context. Typically, conventions do not target a node in isolation but a number of related nodes that form a pattern. The nodes do not have to be immediately related, they could be scattered across the tree.

5) Conventions could be related. For example, it is often the case that two conventions are related within the boundaries of a style guide. Often a given convention serves as an exception for another convention, or relaxes the constraints of another convention.

5.2 Abstract syntax

TODO: shall I add context and convention diagrams Types are still missing

This section describes the abstract syntax of CssCoco. An overview of the abstract syntax is shows in Figure x. A detailed view of each of the .. is presented afterwards.



ConventionSet represents a style guide. It comprises a number of conventions that form coherent guidelines. Attribute `contexts` is a list of Contexts that contain conventions.

Context represents a group of conventions that belong to the same semantic group (e.g. whitespacing, syntax preference, programming style). Attribute `conventions` is a list of Contexts that contain conventions. Attribute `ignored_patterns` is a list of Patterns that are ignored while searching for the target pattern. For example, while searching for violations of semantic conventions, the whitespacing and indentation nodes are ignored.

Convention represents a rule that enforces specific constraints. Attribute `pattern` is the pattern that the convention targets. Attribute `description` is the description of the convention in natural text. This description is displayed to the user when a violation of the convention is discovered.

Pattern represents a description of a node or a combination of related nodes that given convention constraints. Attribute `root` is the top node described in the pattern. Attribute `nodes` is a collection of all nodes described by the pattern. Attribute `relations` is a collection of relationships between the nodes used in the pattern.

SequencePattern represents a special type pattern in which nodes are allowed to be only siblings. Attribute `root` is the top node described in the pattern. Attribute `nodes` is a collection of all nodes described by the pattern. Attribute `relations` is a collection of relationships between the nodes used in the pattern.

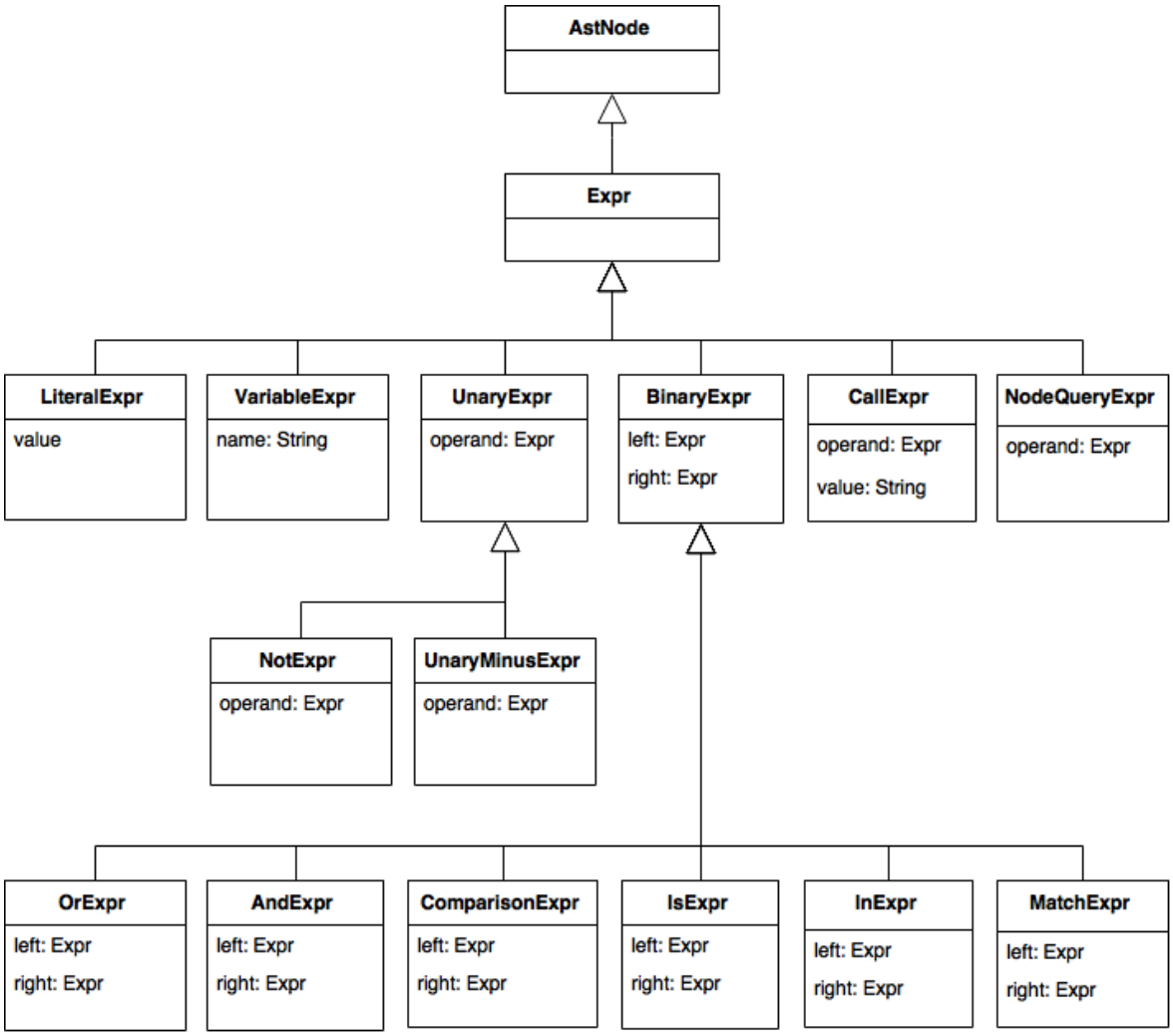
Node represents a description of a non-whitespace node used in a Pattern. Attribute `descriptor` contains information about the type of the described node. Attribute `constraint` is an expression that designates additional constraints applied to the node. Attribute `identifier` is a given string that can be used as a reference to the matched node.

WhitespaceNode represents a description of a whitespace node that references space, newline, indentation symbols. Attribute `descriptor` contains information about the type of the described node. Attribute `repeater` is an optional constraint that specific the number of times a whitespace node can appear consecutively. Repeaters are useful to express conventions that do not specify exact quantities of whitespace symbols. For example, the convention "put at least one blank line between rules" sets a lower limit of the number of blank lines, but not an upper limit.

WhitespaceVariation represents a description of acceptable whitespace sequences. Whitespace variations are allowed to appear as operands of BeforeExpr, AfterExpr and BetweenExpr. Attribute `descriptor` contains information about the type of the described node. Attribute `constraint` is an expression that designates additional constraints applied to the node. Attribute `identifier` is a given string that can be used as a reference to the matched node.

NodeDescriptor represents a description of the type of the Node. Conventions refer to nodes using their CSS type, their function in the CSS program or their concrete syntax representation. For example, the code "#E6E6E6" can be addressed both as a hexadecimal value and as a color. Attribute `func` is a lambda function that determines whether a node meets the type constraint. This function is used to select match candidates while searching the CSS tree for violation patterns. In this way the additional constraints imposed by the Node are evaluated only for the candidates.

NodeRelation represents a relation between two Nodes. Specializations of node relation are previous sibling, next sibling, parent and ancestor relations. Attribute `target_node` designates the Node targeted by the relation.



LiteralExpr represents an expression containing a literal value. Attribute `value` is the value of the literal expression.

VariableExpr represents a reference to a matched node. Attribute `name` is the identifier used to reference the node.

UnaryExpr represents expressions with a single operand. Attribute `operand` is operand of the expression.

NotExpr represents logical negation expression.

UnaryMinusExpr represents unary minus expression.

BinaryExpr represents expressions with a two operands.

Attributes `left` and `right` represent the first and second operands, respectively.

OrExpr represents logical disjunction expression.

AndExpr represents logical conjunction expression.

ComparisonExpr represents expression that compares two operands.

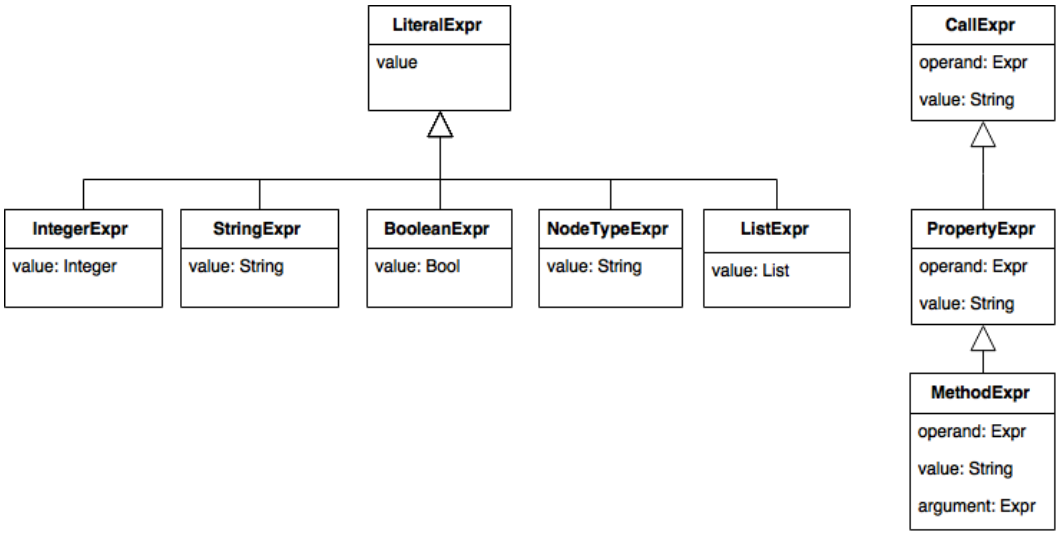
IsExpr represents expression that checks whether the first operand is of the given type, specified by the second operand.

InExpr represents expression that checks whether the first operand is present in a list of values, specified by the second operand.

MatchExpr represents expression that checks whether the first operand matches a regular expression, specified by the second operand.

CallExpr represents expression that invokes a API property or method of the operand. Attribute `operand` is the operand of the expression. Attribute `value` is the name of the API property or method that is invoked.

NodeQueryExpr represents expression that queries node context. Attribute `operand` is the node used as a reference point for the query.



IntegerExpr represents expression containing a integer value.

StringExpr represents expression containing a string value.

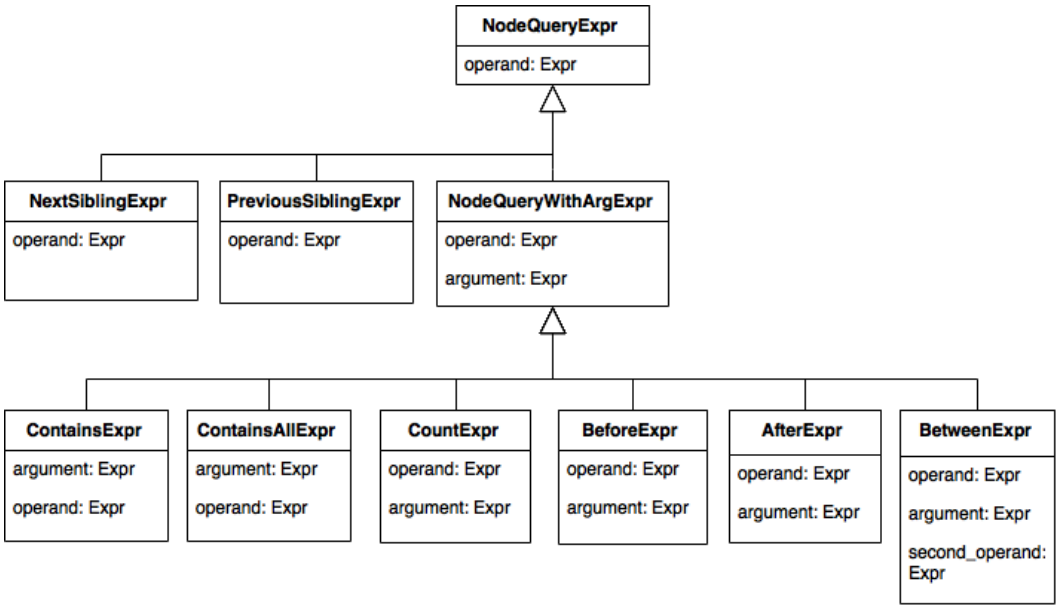
BooleanExpr represents expression containing a boolean value.

ListExpr represents expression containing a list value. The elements of the list are of type LiteralExpr.

NodeTypeExpr represents expression containing a string value that describes node type.

PropertyExpr represents an expression that returns the value of a property of the operand node. Attribute `operand` represents the node targeted by the expression. Attribute `value` holds the name of the property that is accessed.

MethodExpr represents an expression that returns invokes a method of the operand node. Attribute `argument` represents argument passed to the invoked method.



NextSiblingExpr represents expression that returns the following sibling of the operand node.

PreviousSiblingExpr represents expression that returns the previous sibling of the operand node.

NodeQueryWithArgExpr represents expression that queries node context and uses additional constraints for the query. Attribute `argument` represents the additional constraints used by the query.

ContainsExpr represents an expression that checks whether the operand node contains a node that matches given constraints.

ContainsAllExpr represents an expression that checks whether the operand node contains nodes that match given constraints.

CountExpr represents an expression that counts the number of ancestor nodes of the operand that match a given constraint.

BeforeExpr represents an expression that checks whether a given whitespace variation appears before the operand node.

AfterExpr represents an expression that checks whether a given whitespace variation appears after the operand node.

BetweenExpr represents an expression that checks whether a given whitespace variation appears between the two operand nodes.

5.3 Concrete syntax

This section contains the concrete syntax of the designed DSL. Below are presented the grammar rules accompanied by the mapping to the abstract syntax of the language.

stylesheet represents a style guide.

Abstract Syntax Mapping: ast.ConventionSet.

```
stylesheet : context* ;
```

context represents a group of logically related conventions. A single style guide can comprise a number of conventions that enforce various constraints, e.g. whitespacing, syntax preference, program style. Such categories refer to different types of nodes and require ignoring certain patterns.

Abstract Syntax Mapping: ast.Context.

```
context : Identifier '{' convention* '}' ;
```

convention represents a single rule in the style guide. Conventions are typically expressed by directly stating what is disallowed or describing a condition that if met, requires additional constraints. The former way of expressing conventions are represented by the `forbid` conventions. The latter approach uses the structure `find ... require ...`. To break down complex disallowing conventions, the structure `find ... forbid ...` has been introduced. It aims at improving readability of conventions.

Abstract Syntax Mapping: ast.Convention.

```
convention : 'forbid' pattern 'message' String
           | 'find' pattern ('require'|'forbid') logic_expr 'message' String
           ;
```

pattern represents a pattern of nodes and their relations. It can describe a horizontal sequence of sibling nodes and or a vertical pattern of nested nodes. Also, it can describe pairs of elements.

Abstract Syntax Mapping: ast.Pattern.

```
pattern : node_declaration (('in'|'next-to') node_declaration)*
        | fork ('in' node_declaration)*
        ;
fork : '(' node_declaration (',' node_declaration)+ ')' ;
node_declaration : (Identifier '=')? semantic_node ;
```

semantic_node represents a non-whitespace node. It describes the type of the node and its additional constraints.

Abstract Syntax Mapping: ast.Node.

```
semantic_node : node_type ('{' logic_expr '}')? ;
```

whitespace_variation represents a sequence of whitespace nodes. They differ from the rest of the nodes types because the user can specify how many times they need to be consecutively repeated.

Abstract Syntax Mapping: ast.WhitespaceVariation, ast.WhitespaceNode, ast.Repeater.

```
whitespace_variation : whitespace_node ('or' whitespace_node)* ;
whitespace_node : Identifier ('{' repeater '}')? ;
repeater : Integer ',' Integer? | (',')? Integer ;
```

logic_expr represents expressions that perform logic operations and glue arithmetic and type expressions.

Abstract Syntax Mapping: ast.NotExpr, ast.AndExpr, ast.OrExpr and all arithmetic_expression and type_expression mappings.

```
logic_expr : '(' logic_expr ')'
           | 'not' logic_expr
           | logic_expr 'and' logic_expr
           | logic_expr 'or' logic_expr
           | type_expr
           | arithmetic_expr
           ;
```

type_expr represents expressions that ensure node type and perform node queries of whitespace nodes. They are located in a separate parser rule because they interpret Identifiers as node type expressions instead of a API calls.

Abstract Syntax Mapping: ast.IsExpr, ast.BeforeExpr, ast.AfterExpr, ast.BetweenExpr.

```
type_expr : arithmetic_expr operator='is' type_=Identifier
          | whitespace_variation ('before' | 'after') type_operand
          | whitespace_variation 'between' type_operand 'and' type_operand
          ;
type_operand : Identifier | semantic_node ;
```

arithmetic_expr represents arithmetic, comparison, set membership and regex expressions. These are located in a separate parser rule because they interpret identifiers as API calls instead of node type expressions.

Abstract Syntax Mapping: ast.UnaryMinus, ast.UnaryPlus, ast.LessThan, ast.LessThanOrEq, ast.GreaterThan, ast.GreaterThanOrEq, ast.Equal, ast.NotEqual, ast.InExpr, ast.MatchExpr, ast.LiteralExpr.

```
arithmetic_expr : ('-'|'+') arithmetic_expr
                | arithmetic_expr ('<'|'>'|'<='|'>='|'=='|'!=') arithmetic_expr
                | arithmetic_expr ('in'|'not in'|'match'|'not match') arithmetic_expr
                | call_expression
                | element
                ;
element : Boolean | Integer | String | list_ ;
```

call_expr represents an API call expression and also node query expression.

Abstract Syntax Mapping: ast.CallExpr and ast.NodeQueryExpr.

```
call_expr : call_expr '.' call_expr
          | Identifier ('(' (element | semantic_node ) ')')?
          ;
```

Boolean: represents Boolean literal expression.

Abstract Syntax Mapping: ast.BooleanExpr.

```
Boolean : 'true' | 'True' | 'false' | 'False' ;
```

String: represents String literal expression.

Abstract Syntax Mapping: ast.StringExpr.

```
String : "'" (EscapeSequence | ~['])*? "'" ;
EscapeSequence : "\\" "'" ;
```

Integer: represents Integer literal expression.

Abstract Syntax Mapping: ast.IntegerExpr.

```
Integer : (ZeroDigit | NonZeroDigit Digit*) ;
Digit : ZeroDigit | NonZeroDigit ;
NonZeroDigit : [1-9] ;
ZeroDigit : [0] ;
```

list and **list_element** represent the List literal expression.

Abstract Syntax Mapping: ast.ListExpr.

```
list_ : '[' list_element (',' list_element)* ']' ;
list_element : Integer | String | semantic_node ;
Letter : [a-zA-Z] ;
Identifier : (Letter)(Letter|Digit|'_'|'-')* ;
```

type_expression represents the NodeType literal expression.

Abstract Syntax Mapping: ast.NodeType.

```
node_type : '(' node_type ')'
          | 'not' node_type
          | node_type 'and' node_type
          | node_type 'or' node_type
          | Identifier
          ;
```

5.4 Validation

The method chosen for validating the designed domain-specific language is ontological analysis. It is an established way for evaluating software notations []. This approach requires designing a domain-specific ontology and using it as a reference for the ontological analysis. The leading top-level ontology used for ontological analysis is the Bunge-Wand-Weber (BWW) ontology [...]. The designed domain-specific ontology is based on it too.

Why ontologies are helpful in DSL development [http://ceur-ws.org/Vol-395/paper02.pdf]

5.4.1 Ontology design

There is not a formal method for designing ontologies. Formulating concepts is always subjective.

Class **Style Guide** describes the coding practices adopted in the context of a single project, organization, community or language. An individual Style Guide is a composite thing built of Conventions and their relations. Conventions in a Style Guide are interpreted together to form a coherent set of guidelines.

Property **Number of Conventions** indicates the size of the Style Guide.

Class **Convention** is a rule that imposes constraints on the CSS program. It is the building block of Style Guides. An individual Convention is a composite thing that consists of a Context.

Intrinsic Property **Description** explains the meaning of the Convention in natural text.

Mutual Property **Exception** is a relation between Conventions in which a given convention serves as an exception of another Convention.

Mutual Property **Allowance** is a relation between Conventions in which a given convention relaxes the constraints of another Convention.

Class **Context** is a description of a Pattern that the Convention forbids. An individual Context is a composite thing that comprises a number of logically related Constraints.

Property **Ignored Constructs** specifies constructs that are disregarded while searching for a Context.

Class **Constraint** is a restriction that needs to be fulfilled. Different types of Constraints are represented as kinds.

Property **Subject** indicates the operand on which the Constraint operates. Things that can be a subject are the Type, Textual Representation or CSS-specific knowledge of a Construct.

Kind **Comparison Constraint** is a type of Constraint that compares the subject to another value.

Property **Subject**

Property **Value**

Kind **Type Constraint** is a type of Constraint that checks whether the subject is of a given type.

Property **Subject**

Property **Type**

Kind **Textual Form Constraint** is a type of Constraint that imposes restrictions on the textual representation of the subject.

Property **Subject**

Property **Form**

Kind **Set Membership Constraint** is a type of Constraint that requires the subject to be a member of a set.

Property **Subject**

Property **Set**

Kind **Context Constraint** is a type of Constraint that requires the subject to be in a particular context.

Property **Subject**

Property **Context Description**

Class **Violation Log** is the final product of a violations search. An individual Violation Log is a composite thing that contains Violations.

Property **Number of Violations** indicates the size of the Violation Log.

Class **Violation** A Violation occurs when a Pattern that matches the Context of a Convention is found.

Property **Description** explains in natural text what causes the Violation. Typically, the Description is extracted from the Convention that the Violation breaks.

Property **Position in Stylesheet** indicates the location of the Pattern that violates the Convention in the Stylesheet.

Class **Stylesheet** is the CSS code that needs to be checked for compliance with the Style Guide. An instance of Stylesheet is a composite thing that comprises a number of Constructs

Class **Construct** is a part of the Stylesheet. It can refer to nodes in the CSS abstract syntax tree, concrete syntax tree and parse tree. Examples include whitespacing, indentation, comments, colons, delimiters, rulesets, declarations, etc.

Property **Type** identifies the function of a Construct in the program. Examples are strings, attribute values etc.

Property **Textual Representation** is the string of a Construct that appears in the CSS program. Examples are tabs, ;, and .myclass

Property **CSS-specific knowledge** encapsulates properties of nodes specific to the CSS domain. For example, a CSS declaration node can possess knowledge whether it is vendor specific or not.

Mutual Property **Relation** is a relation between two Constructs. For example, two Constructs may be adjacent or nested.

Class **Pattern** is a particular part of the CSS program that matches the description of a Context. An instance of a Pattern is a composite thing built from one or many Constructs and Relations between them.

Property **Number of Constructs** denotes the size of the Pattern.

Event **Search for Violations in Stylesheet** occurs when the developer completes the search for violations in a Stylesheet, a Violation Log is created.

New State **Violation Log** { Violations = value }

Event **Context (of Convention) Discovered** occurs when the Context of a convention is discovered, a Violation is recorded in the Violation Log. The state of the Violation contain its description and position in Stylesheet.

New State **Violation** { Description = value, Position in Stylesheet = value }

TODO: include property and event of the Stylesheet! TODO: include couplings diagram with explanation!

5.4.2 Ontological analysis

You can refer to pretty much anything (websites, blog posts, software) through `misc` type of entry [**?**]:

Chapter 6

Conclusion

This work makes several contributions. 1) 2) It contains a summary of existing CSS coding conventions. 3) It designs a domain specific language that expresses conventions and its interpreter to detect violations automatically.

Bibliography

[GE07] Andreas Gehlert and Werner Esswein. Toward a formal research framework for ontological analyses. *Advanced Engineering Informatics*, 21(2):119–131, 2007.

[GR00] Peter Green and Michael Rosemann. Integrated process modeling: an ontological evaluation. *Information systems*, 25(2):73–87, 2000.

[Gru95] Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International journal of human-computer studies*, 43(5):907–928, 1995.

[Moo09] Daniel L Moody. The physics of notations: toward a scientific basis for constructing visual notations in software engineering. *Software Engineering, IEEE Transactions on*, 35(6):756–779, 2009.

[MS04] Simon K Milton and Barry Smith. Top-level ontology: The problem with naturalism. In *Formal ontology in information systems*, pages 85–94, 2004.

[WW90] Yair Wand and Ron Weber. An ontological model of an information system. *Software Engineering, IEEE Transactions on*, 16(11):1282–1292, 1990.

[WW95] Yair Wand and Ron Weber. On the deep structure of information systems. *Information Systems Journal*, 5(3):203–223, 1995.