

# Detecting Violations of CSS Code Conventions

**Boryana Goncharenko**

[boryana.goncharenko@gmail.com](mailto:boryana.goncharenko@gmail.com)

July 18, 2015, 24 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

<b>Abstract</b>	<b>2</b>
<b>1 Introduction</b>	<b>3</b>
<b>2 Background</b>	<b>4</b>
2.1 Bunge-Wand-Weber ontology . . . . .	4
2.2 Ontological analysis . . . . .	5
<b>3 Evaluating the Need for CSS Code Conventions</b>	<b>6</b>
3.1 Research Method . . . . .	6
3.2 Results . . . . .	6
3.3 Analysis . . . . .	6
<b>4 Discovering Existing CSS Code Conventions</b>	<b>8</b>
4.1 Research Method . . . . .	8
4.2 Results . . . . .	8
4.3 Analysis . . . . .	9
<b>5 Expressing CSS Code Conventions</b>	<b>10</b>
5.1 Analysis of conventions corpus . . . . .	10
5.2 Abstract syntax . . . . .	11
5.3 Concrete syntax . . . . .	15
5.4 Validation . . . . .	18
5.4.1 Ontology design . . . . .	18
5.4.2 Ontological analysis . . . . .	21
<b>6 Conclusion</b>	<b>23</b>
<b>Bibliography</b>	<b>24</b>

# 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 illustrated 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

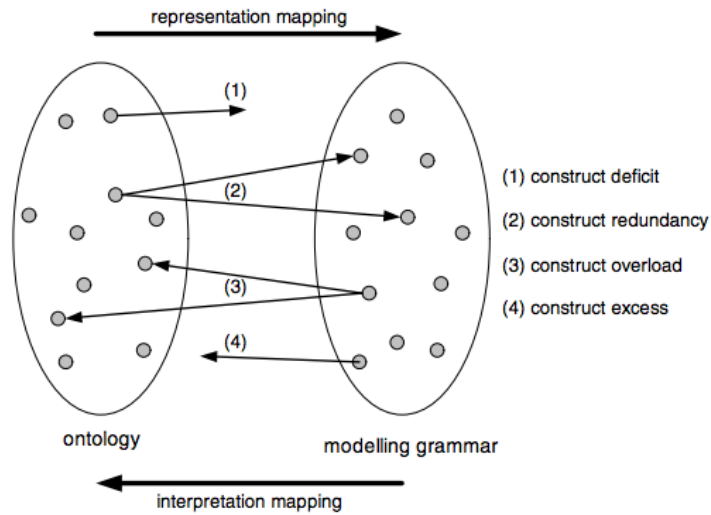
Ontological construct	Explanation
Thing	The elementary unit in the BWW ontological model. The real world is made up of things. A composite thing may be made up of other things (composite or primitive).
Properties	Things possess properties. A property is modeled via a function that maps the thing into 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 more 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 underpinning of ontological analysis is that modeling grammars are incomplete if they are not able to represent what exists in reality [GR00]. Furthermore, the analysis requires 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

### 3.1 Research Method

Despite the new features added in the second [BCHL11] and third [Ete11] versions of CSS, the language has obvious limitations, for example, lack of variables. A number of preprocessors have evolved to tackle the downsides of CSS. Solutions such as SASS [CWE06], LESS [SSP+09] and Stylus [Hol15] offer enhanced or even different syntax and translate it to CSS. Preprocessors are not only ubiquitously recommended, but also widely adopted in practice. The presence of such solutions poses the question whether conventions for CSS are required at all. If nowadays CSS is generated and not handcrafted, there is not need for conventions for CSS. There is need for conventions for the preprocessor languages.

To determine whether CSS is still handcrafted, all commits to open source repositories hosted on GitHub for the period Jan-Apr 2015 have been analyzed. To differentiate between plain CSS and preprocessor code, the extensions of all files in the commits were inspected. In case 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 plain CSS. Since the main objective of the search is to determine maintenance, only files that were modified are taken into consideration. Files that were added are excluded from the results, since developers often add third-party CSS libraries to their repositories.

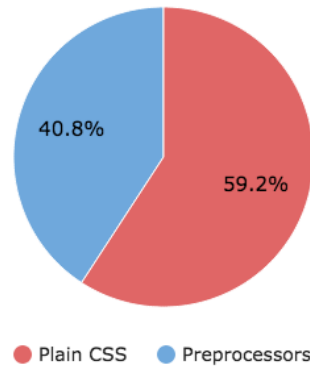
### 3.2 Results

A total of 1,589,713 commits to 1,311,654 public repositories have been analyzed. X of these were not processed due to errors. Typical errors contain repositories that have been turned private and... The number of commits that maintain any form of CSS is X. Figure 1 summarizes the findings.

### 3.3 Analysis

There are a number of limitations that need to be considered before interpreting the results of the experiment. Firstly, the search is conducted on a single hosting platform - GitHub. That said, currently GitHub reports having over 10 million users and 24 million repositories [Git], which makes it the largest code host in the world [GVSZ14]. Secondly, the search is narrowed to the publicly available repositories. Thus, it lies on the premise that there is not a significant difference between the public and private repositories hosted on the platform. Thirdly, the experiment detects only the three most popular preprocessor extensions and omits other preprocessors. It is possible that a number of custom preprocessors are used in practice. However, it is assumed that the number of such commits would not increase the number of total CSS commits to the extend at which the percentage of plain CSS commits is significantly diminished.

Figure 3.1: Results



Having the above limitations in mind, the experiment provides evidence to conclude that despite the popularity of preprocessors, plain CSS is still handcrafted in the beginning of 2015.



## Chapter 4

# Discovering Existing CSS Code Conventions

### 4.1 Research Method

The primary organization responsible for the specification of CSS has not published an official CSS style guide. As a result, the CSS community has produced a pool of coding conventions, best practices, guidelines and recommendations.

To discover existing CSS code conventions, searches using two search engines <http://duckduckgo.com> and <http://google.com> have been conducted and the first 100 results of each search have been analyzed. From each result only conventions about plain CSS have been taken into account. Conventions that refer to CSS preprocessors have been ignored. In case the result contains links to other style guides, these references have been considered as results and analyzed separately.

During the process of discovering conventions, a number of issues have been observed. First, some of the conventions did not provide sufficient information to be applied in practice. Such an example is the convention “dont use CSS hacks try a different approach first” while the style guide does not define the notion of CSS hacks. Overgeneralized conventions have been omitted from the results.

Another part of the discovered conventions introduced a discrepancy between their description in natural language and the provided code 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. In such cases the convention is interpreted as described by the code example.

In some cases, when conventions are not supported with code examples, their description remains open for interpretation. 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. All possible interpretations of ambiguous conventions were registered as separate conventions.

There are conventions that are not explicitly stated, but could only be inferred by the other rules. For example, the convention “you can put long values on multiple lines” implies that values should appear on one line. Implicit conventions were registered as explicit conventions.

### 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.

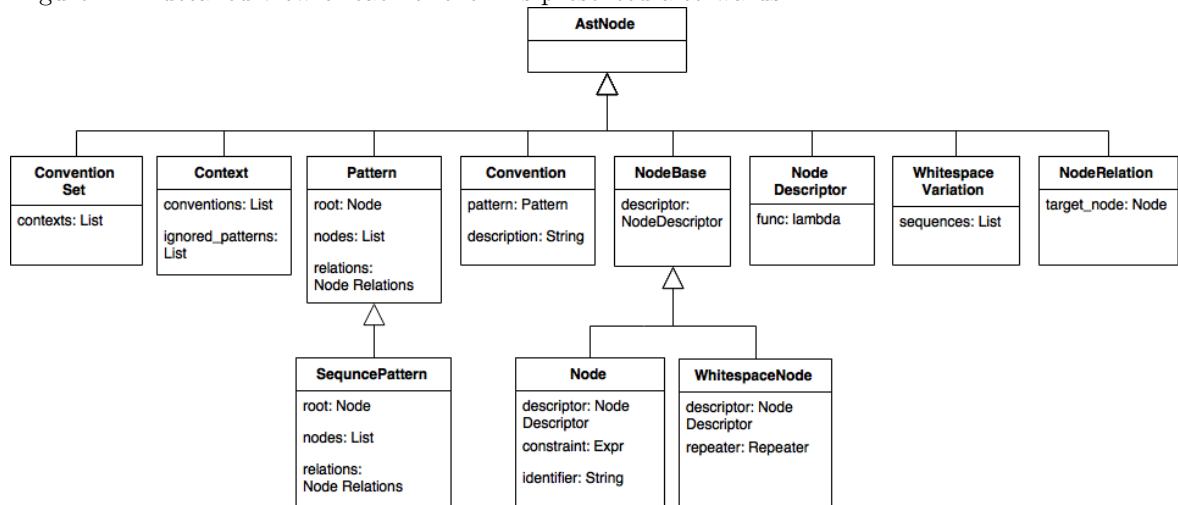
In this way, all conventions have been analyzed and as a result, the following set of conclusions have been made:

- 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 shown 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. whitespace, 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 whitespace 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.

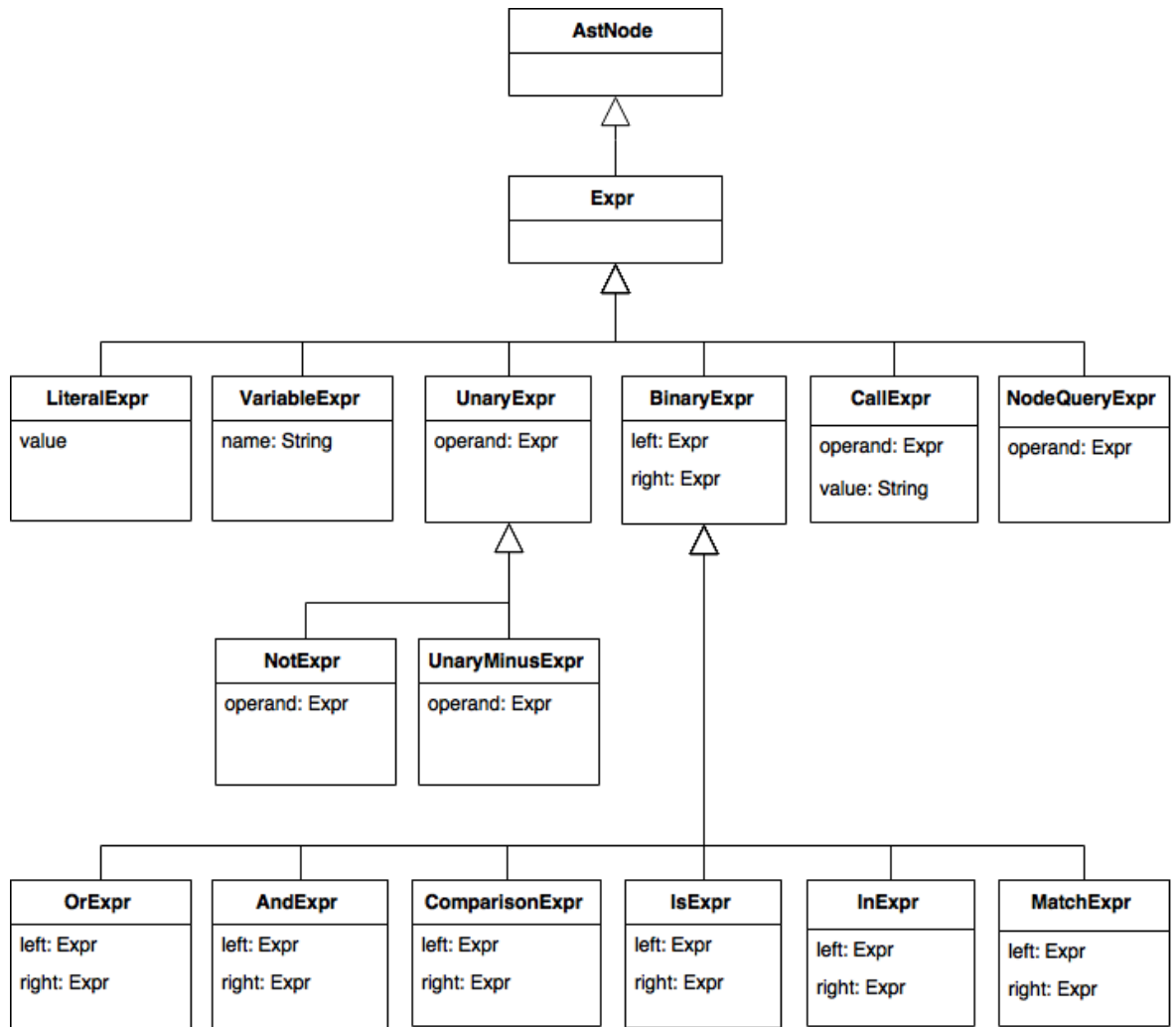
**Node** represents a description of a non-whitespace node used in a Pattern. Attribute **descriptor** contains information about the type of the described node. Attribute **constraint** is an expression that designates additional constraints applied to the node. Attribute **identifier** is a given string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

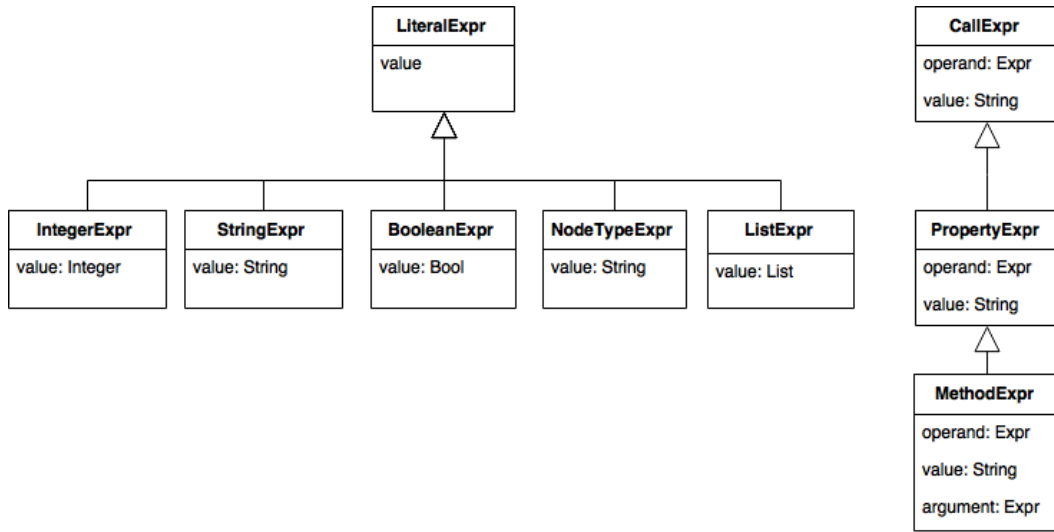
**IsExpr** represents expression that checks whether the first operand is of the given type, specified by the second operand.

**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.

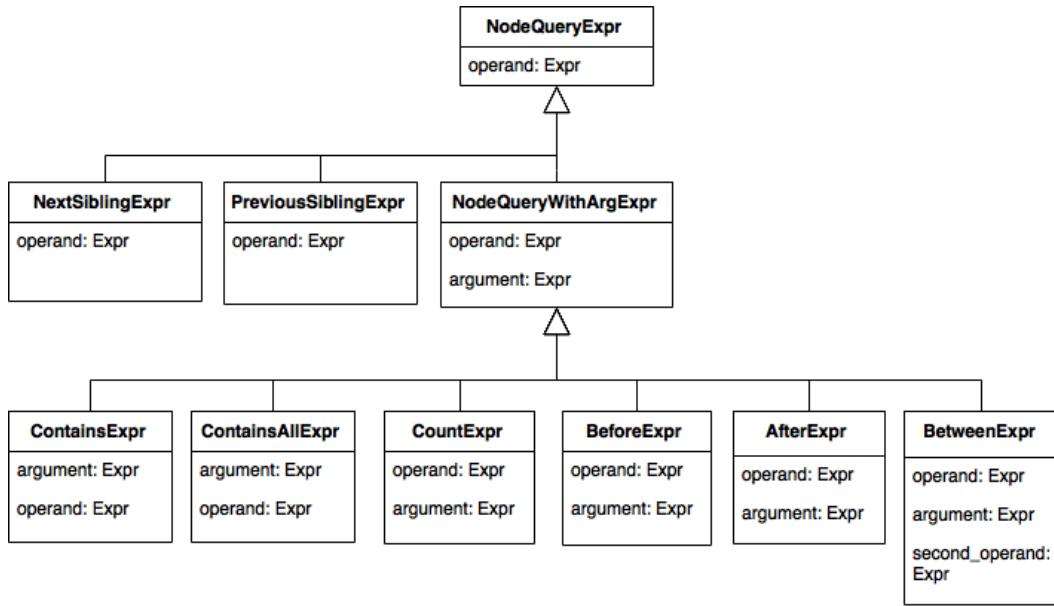
**BooleanExpr** represents expression containing a boolean value.

**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. whitespace, 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|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, since it is a widely-accepted way for evaluating software notations [OHS02, GR00, Moo09, PW97, WZ96]. The particular approach used for conducting ontological analysis consists of several steps. First, a domain-specific ontology is designed. Second, the ontology is used as a reference point for the interpretation and representation mappings. Third, emerged anomalies are analyzed and conclusion about the quality of the notation is made.

### 5.4.1 Ontology design

The first stage of validation requires designing a domain-specific ontology. The specific domain of the developed ontology is limited to detecting violations of CSS code conventions. In other words, the designed ontology tries to capture only the concepts and their relations, that exist when an agent searches a CSS program for violations of given set of code conventions.

The designed domain-specific ontology is based on the BWW top-level ontology [WW90], i.e. it uses the high-level categories of the BWW ontology to describe the objects, concepts and entities in the specific domain. The rationale behind the decision to use BWW ontology is that it has been the leading ontology used for ontological analysis [Moo09]. The main ontological constructs used in the BWW ontology are listed in section 2.1.

Following is a list with the main concepts discovered in the domain along with their descriptions. The used BWW concepts are written in *italics* and the domain-specific concepts are written in **bold**.

*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 }

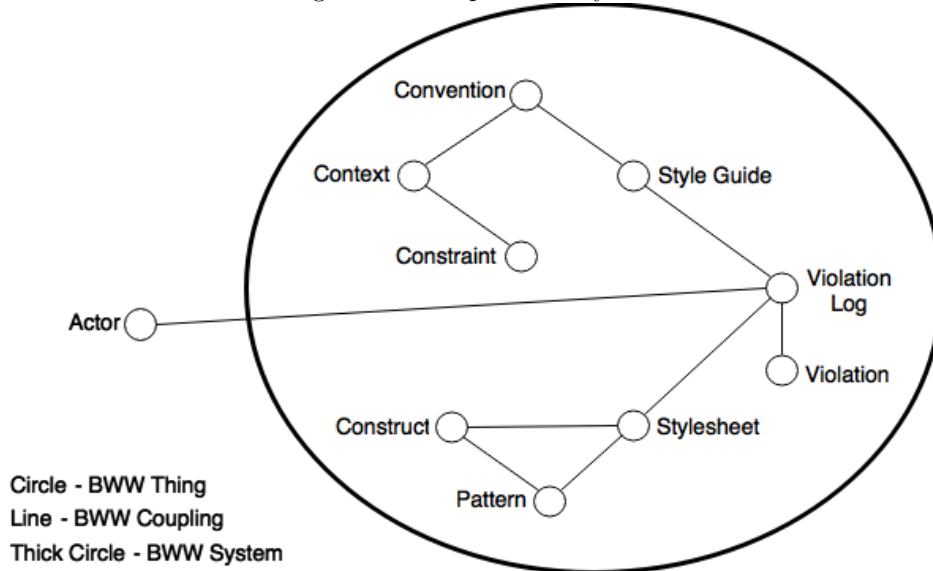
TODO: include property and event of the Stylesheet! TODO: include couplings diagram with explanation!

The descriptions provided in the listing above often state that an instance of a class is a composite thing that consists of other things. To provide a better understanding of the relations between the entities defined in the ontology, the concepts are also expressed using the Backus-Naur Form (BNF) notation.

STYLE_GUIDE	::= CONVENTIONS
CONVENTIONS	::= CONVENTIONS CONVENTION   CONVENTION   EPSILON
CONVENTION	::= CONTEXT
CONTEXT	::= CONSTRAINTS
CONSTRAINTS	::= CONSTRAINTS CONSTRAINT   CONSTRAINT
CONSTRAINT	::= COMPARISON_CONSTRAINT   TYPE_CONSTRAINT   TEXT_CONSTRAINT   SET_CONSTRAINT   CONTEXT_CONSTRAINT
VIOLATION_LOG	::= VIOLATIONS
VIOLATIONS	::= VIOLATIONS VIOLATION   VIOLATION   EPSILON
STYLESHEET	::= CONSTRUCTS
CONSTRUCTS	::= CONSTRUCTS CONSTRUCT   CONSTRUCT   EPSILON
PATTERN	::= CONSTRUCTS CONSTRUCT_RELATIONS

While the above listing presents the composition of things, it does not illustrate how things interact with each other. To provide a better understanding of the dynamics between things defined in the ontology, a diagram of the system is presented in Figure 5.1.

Figure 5.1: Graph of the system



According to the theory, a coupling occurs when the existence of a given thing affects the history of another thing and, in turn, history is defined as the chronological ordered states that a thing traverses [WW90]. Thus, in the domain-specific ontology a coupling exist between the Style Guide and the Convention things, because the existence of a Convention alters the state of the Style Guide. Similarly, a Context changes the state of a Convention and a Constraint affects the state of a Context.

There are also couplings between Construct, Pattern and Stylesheet things. Both Stylesheet and Pattern are composed of Constructs, and thus affected by their existence. Since a Pattern is a specific occurrence of a combination of Constructs, it is also coupled to Stylesheet.

A Violation is coupled to a Violation Log, since the presence of a new a Violation alters the state of the log. Further, a Violation Log contains information about the violations of a particular Style Guide that occur in a specific Stylesheet. In this sense, a Violation Log is a function of a Style Guide and a Stylesheet and it is coupled to both things.

There are two external events that an Actor can initiate: searching for violations and discovering violations. Both of these events express couplings, because they change the state of the Violation Log and the Violation, respectively. Note that an Actor can create any of the things that appear in Figure 5.1. In practice, the Actor can introduce additional Conventions, change the Style Guide, alter the existing Constructs and rewrite the Stylesheet. However, these events are considered outside the domain of the designed ontology.

## 5.4.2 Ontological analysis

### Representation Mapping

Table 5.1: Representation mapping

Ontological concepts	Semantic constructs
Style Guide	Convention Set
Number of Conventions of Style Guide	Conventions of Convention Set
Convention	Convention
Description of Convention	Description of Convention
Exception of Conventions	Exceptions of Convention
Allowance of Conventions	Relaxing Conventions of Convention
Context	Pattern descriptor, Node descriptor, Node Relation
Ignored Constructs	Convention Context
Constraint	Expression
Subject of Constraint	API Call, Literal Expression, Variable Expression
Comparison Constraint	Comparison Expression
Type Constraint	Is Expression
Textual Form Constraint	Match Expression
Set Membership Constraint	Set Membership Expression
Context Constraint	Context Query Expression
Violation Log	Violation Log
Number of Violations	Violations of Violation Log
Violation	Violation
Description of Violation	Description of Violation
Position in Stylesheet of Violation	Position of Violation
Stylesheet	Tree
Construct	Node
Type of Construct	Search Categories of Node
Textual Representation of Construct	String of Node
CSS-specific knowledge	API of Node
Relation between Constructs	Node Relation
Pattern	Node, Node Relation
Number of Constructs of Pattern	Node Descriptors of Pattern Descriptor

Table 5.2: Interpretation mapping

Semantic constructs	Ontological concepts
Convention Set	Style Guide
Conventions of Convention Set	Number of Conventions of Style Guide
Convention Context	Ignored Constructs of a Context
Convention	Convention
Description property of Convention	Description of Convention
Exceptions of Convention	Exception of Conventions
Relaxing Conventions of Convention	Allowance of Conventions
Pattern Descriptor, Node Descriptor, Node Relation	Context
Node Descriptors of Pattern Descriptor	Number of Constructs of Pattern
Expression, Logical Expression	Constraint
Comparison Expression	Comparison Constraint
Is Expression	Type Constraint
Match Expression	Textual Form Constraint
Set Membership Expression	Set Membership Constraint
Context Query Expression	Context Constraint
API Call, Literal Expression, Variable Expression	Constraint Subject
Node	Construct
Search Categories of Node	Type of Construct
String of Node	Textual Representation of Construct
API of Node	CSS-specific knowledge of Construct
Node, Node Relation	Pattern
Tree	Stylesheet
Violations Log	Violations Log
Violations of Violations Log	Number of Violations
Violation	Violation
Description of Violation	Description of Violation
Position of Violation	Position in Stylesheet of Violation

## 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

- [BÇHL11] Bert Bos, Tantek Çelik, Ian Hickson, and Hakon Wium Lie. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. *W3C Recommendation*, June 2011. <http://www.w3.org/TR/2011/REC-CSS2-20110607>.
- [CWE06] Hampton Catlin, Natalie Weizenbaum, and Chris Eppstein. SASS: Syntactically Awesome Style Sheets, 2006. <http://sass-lang.com>.
- [Ete11] Elika J. Etemad. Cascading Style Sheets (CSS) Snapshot 2010. *W3C Working Group Note*, May 2011. <http://www.w3.org/TR/2011/NOTE-css-2010-20110512/>.
- [GE07] Andreas Gehlert and Werner Esswein. Toward a formal research framework for ontological analyses. *Advanced Engineering Informatics*, 21(2):119–131, 2007.
- [Git] GitHub. <https://github.com/about/press>.
- [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.
- [GVSZ14] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 384–387. ACM, 2014.
- [Hol15] T. J. Holowaychuk. Stylus, 2015. <https://learnboost.github.io/stylus>.
- [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.
- [OHS02] Andreas L Opdahl and Brian Henderson-Sellers. Ontological evaluation of the uml using the bunge–wand–weber model. *Software and systems modeling*, 1(1):43–67, 2002.
- [PW97] Jeffrey Parsons and Yair Wand. Using objects for systems analysis. *Communications of the ACM*, 40(12):104–110, 1997.
- [SSP<sup>+</sup>09] Alexis Sellier, Jon Schlinkert, Luke Page, Marcus Bointon, Mria Juroviov, Matthew Dean, and Max Mikhailov. Less, 2009. <http://lesscss.org>.
- [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.
- [WZ96] Ron Weber and Yanchun Zhang. An analytical evaluation of niam’s grammar for conceptual schema diagrams. *Information Systems Journal*, 6(2):147–170, 1996.