Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Abschlussarbeit

Alexander Holtkamp

Entwurf und Implementation eines B.E.M. Plugins

für IntelliJ IDEA

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Holtkamp

Entwurf und Implementation eine B.E.M. Plugins

für IntelliJ IDEA

Abschlussarbeit eingereicht im Rahmen Studium

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Olaf Zukunft
Zweitgutachter : Prof. Dr. Michael Neitzke

Abgegeben am 17. Mai 2016

Alexander Holtkamp

Thema der Arbeit

Entwurf und Implementation eines B.E.M. Plugins für IntelliJ IDEA

Stichworte

HTML, CSS, Block, Element, Modifier, BEM, Benennung, Muster, Frontend, JetBrains, Intellij, IDEA, Plugin, Parser, Lexer, Sprache, Kaskade, Selektor, OOP, Architektur

Kurzzusammenfassung

Das „Block Element Modifier“-Pattern ist ein Benennungsmuster für Klassennamen in Cascading Style Sheets (CSS), welches sich speziell für größere Frontend-Projekte eignet. Diese Bachelorarbeit stellt das Muster und seinen Nutzen für die Frontend-Architektur vor. Aus der Sicht von möglichen Anwendern wird außerdem beschrieben, wie man in der Entwicklungsumgebung IntelliJ IDEA bei der richtigen Nutzung des B.E.M. Patterns unterstützt werden kann. Die so entstandenen User-Stories bilden dann die Grundlage für den Software-Entwurf, die Implementation und entsprechende Tests für ein B.E.M. Plugin. Das funktionsfähige Ergebnis liegt der Arbeit bei und wird als Projekt fortgesetzt.

Alexander Holtkamp

Title of the paper

Design and Implementation of a B.E.M. Plugin for IntelliJ IDEA

Keywords

HTML, CSS, block, element, modifier, BEM, naming, pattern, frontend, JetBrains, Intellij, IDEA, plugin, parser, lexer, language, cascade, selector, OOP, architecture

Abstract

The "block element modifier" pattern is a naming pattern for class names in cascading style sheets (CSS) which is especially suitable for larger frontend projects. This bachelor thesis explains the pattern und its benefits for the frontend architecture. In addition to this from the point of view of possible users it is described how the development environment IntelliJ IDEA could be able to support the proper usage of the pattern. The user-stories that have been created that way build the foundation of the software design, the implementation and fittable tests for a B.E.M. plugin. The result is attached to this bachelor thesis and will be continued as a project.

Inhaltsverzeichnis

1 Einleitung

Die tägliche Arbeit von Software-Entwicklern wird immer mehr durch Entwicklungsumgebungen unterstützt. Gerade bei der Java-Programmierung ist die Unterstützung hierbei schon sehr weit fortgeschritten:

So wird beispielsweise vor möglichen Fehlern gewarnt, das Umwandeln von Code in eine äquivalente, aber verbesserte Form vorgeschlagen, oder es werden ganze Code-Blöcke generiert.

Um dies zu ermöglichen, bieten viele Entwicklungsumgebungen eine Schnittstelle an, mit der sich ihre Funktionalität erweitern lässt. Da außerhalb der Java-Programmierung die Unterstützung noch nicht so weit fortgeschritten ist, schafft dies die Möglichkeit, auch für andere Sprachen sinnvolle Erweiterungen vorzunehmen und diese beispielsweise als Plugin bereitzustellen.

1.1 Ziel der Arbeit

Im Zuge dieser Arbeit soll für die Sprache CSS das B.E.M. Pattern inklusive der dafür erforderlichen technischen Grundlagen beschreiben und darauf aufbauend ein funktionsfähiges und getestetes Plugin für die Entwicklungsumgebung IntelliJ IDEA bereitgestellt werden, welches bei der Verwendung des Patterns unterstützen soll.

1.2 Vorgehensweise

Um dies zu erreichen, werden zunächst mit Kapitel 2 alle Begriffe und Methoden erklärt, auf die die darauf folgenden Kapitel aufbauen:

Ich stelle die im World-Wide-Web weit verbreitete Auszeichnungssprache HTML und den daraus vom Browser erzeugten DOM vor, auf den sich dann sogenannte CSS-Regeln beziehen können, um das Aussehen einer Webseite zu beschreiben.

Als weitere Technologie stelle ich die objektorientierte Programmierung am Beispiel der später für das Plugin verwendeten Programmiersprache Java vor.

Nach Vorstellung der Technologien wird die Definition des B.E.M. Patterns als reines Namenspattern gezeigt, für das das Plugin Unterstützung anbieten soll. Daraufhin folgen Kapitel, die sich mit der Entwicklungsumgebung IntelliJ IDEA befassen, für die das Plugin zur Verfügung gestellt werden soll. Hier werden grundlegende Begriffe der Plugin-API erklärt und mögliche Erweiterungspunkte gezeigt, an denen das Plugin sich einhängen könnte.

Zum Abschluss der Grundlagen gibt es noch ein Kapitel über die „Erweiterte Backus-Naur-Form“, welche Anwendung im Implementationsteil finden wird.

Kapitel 3 befasst sich zunächst mit der Analyse des vorgestellten Patterns und stellt Zusammenhänge zur objektorientierten Programmierung in Java her, anhand derer dann gezeigt werden soll, dass es sich nicht nur um ein Namens- sondern auch um ein Strukturmuster handelt. Eine Vorstellung von möglichen Erweiterungen der in B.E.M. definierten Regeln, soll zeigen, wie das B.E.M. Pattern weitere Prinzipien der Software-Entwicklung unterstützen kann.

Der zweite Teil des Kapitels befasst sich mit dem Design des B.E.M. Plugins und stellt die einzelnen Funktionalitäten, die implementiert werden sollen, in Form von User-Stories vor, welche gewöhnlich in der agilen Softwareentwicklung angewandt werden.

Im Unterkapitel Anwendung der Plugin-API wird beschrieben, mit welchen Features der IntelliJ Plugin-API die einzelnen User-Stories umgesetzt werden sollen.

Die eigentliche Umsetzung wird dann in Kapitel 4 erklärt. Da der dokumentierte Quellcode des Plugins dieser Arbeit beiliegt, werden hier nur besondere Aspekte oder Probleme bei der Implementierung geschildert und die finale Paketstruktur gezeigt.

Um die Richtigkeit der Implementierung sicherzustellen, werden danach Methoden vorgestellt, mit denen die einzelnen Funktionalitäten getestet werden.

Abschließend bewerte ich das Vorgehen bei der Plugin-Entwicklung, fasse die Ergebnisse kurz zusammen und gebe einen Ausblick auf mögliche neue Anwendungsfälle für das Plugin.

1.3 Danksagungen

Ich möchte mich an dieser Stelle insbesondere bei meinem Vater für die finanzielle und moralische Unterstützung während meines Studiums bedanken.

Besonderer Dank geht auch an meinen Arbeitgeber CoreMedia, der mir viel Freiraum bei der Einteilung meiner Arbeitszeit gegeben hat, sodass ich die Bachelorarbeit neben dem Beruf fertigstellen konnte.

Ebenfalls bedanken möchte ich mich bei meinem betreuenden Professor Dr. Olaf Zukunft für viele wertvolle Tipps und die Geduld, die er mir entgegengebracht hat, wenn ein neuer Zwischenstand einmal auf sich warten ließ.

Zu guter Letzt möchte ich mich bei meinen Freunden bedanken, die mir Ablenkung beschert haben, wenn ich sie brauchte, und mich immer wieder neu motiviert haben, wenn ich einmal den Antrieb verloren habe.

2 Grundlagen

In diesem Kapitel werden die grundlegenden Begriffe und Konzepte vorgestellt, die zum Verständnis dieser Arbeit beitragen. Ich gehe davon aus, dass Kenntnisse über die Java Entwicklung vorhanden sind, da hier lediglich eine Teilmenge der Begriffe definiert wird.

2.1 Frontend-Entwicklung

Die Begriffe Frontend und Backend (manchmal auch Front-End und Back-End geschrieben) bedeuten wörtlich übersetzt vorderes Ende und hinteres Ende. Hiermit ist eine mögliche Schichtentrennung innerhalb der Software gemeint, der sich eine Teilmenge der Einzelkomponenten zuordnen lässt, aus denen sich die Software zusammensetzt.

Während das Frontend in der Regel näher am Benutzer der Software verankert ist, sitzt das Back-End entsprechend näher am System. Häufig ist hiermit einfach die Benutzungsoberfläche gemeint, die eine Software bereitstellt. [1]

Die folgenden Unterkapitel stellen zwei wichtige Technologien vor, die sich für Benutzungsoberflächen im World-Wide-Web etabliert haben.

2.1.1 HTML

Die **H**yper**t**ext **M**arkup **L**anguage ist eine Auszeichnungssprache zur Strukturierung von digitalen Inhalten. Die in dieser Sprache erfassten HTML-Dokumente sind die Basis des World-Wide-Web und werden von sogenannten Web Browsern dargestellt.

HTML wird - ähnlich wie XML - mit sogenannten HTML-Tags beschrieben. Tags werden mit Hilfe von spitzen Klammern in der Sprache identifiziert und bestehen meistens aus einem

[1] (Fischer und Hofer 2008)

einleitenden Start-Tag, das kennzeichnet, dass das Element hier beginnt, dem Inhalt des Tags und einem schließenden End-Tag, welches das Ende kennzeichnet.

Der Inhalt kann aus weiteren HTML-Elementen oder aus einfachem Text bestehen. Durch diese Notation entsteht eine Verschachtelung von Auszeichnungen um den jeweiligen Text, welche ihn semantisch anreichert.

Ein Tag erhält als erste Information einen eindeutigen Bezeichner, der den Typen des Tags festlegt. Abhängig von dem Typen ist dann die Festlegung einer Reihe von Attributen möglich. Während einige Typen, wie das ***id*** oder ***class*** Attribut von allen Typen unterstützt werden, sind bestimmte Attribute nur für bestimmte Tag-Typen vorgesehen (z.B. ***href***).

Im Folgenden wird das ***id*** Attribut eines HTML-Elements als seine ID bezeichnet und die einzelnen, durch Leerzeichen separierten Strings, die sich im ***class*** Attribut eines HTML-Elements befinden, werden als CSS-Klasse bezeichnet (vgl. Kapitel 2.1.3). [2]

2.1.2 DOM

Das **D**ocument **O**bject **M**odel oder kurz DOM stellt eine plattform-neutrale Schnittstelle für Events und Knotenbäume dar. Browser benutzen dieses Interface um den Zugriff auf HTML oder XML Dokumente zu ermöglichen.

Während Events nicht Gegenstand dieser Ausarbeitung sein sollen, sind die Begriffe, die sich aus der Beziehung einzelner DOM-Knoten zueinander ergeben, von großer Bedeutung.

Bei der Beschreibung der Beziehungen der DOM-Knoten untereinander betrachte ich den DOM-Knotenbaum als gewurzelten Baum, dessen Kanten von der Wurzel ausgehen, mit dem HTML-Element als Wurzel.

Hierdurch lassen sich folgende Knotenmengen bilden:

- **Kind-Knoten**
  Ein Knoten C ist ein Kind-Knoten eines Knotens P, wenn es eine Kante E von P nach C gibt.

- **Eltern-Knoten**
  Ein Knoten P ist ein Eltern-Knoten von einem Knoten C, wenn es eine Kante E von P nach C gibt.

[2] (Smith 2013)

- **Geschwister-Knoten**
  Ein Knoten C1 ist ein Geschwister-Knoten von C2, wenn beide denselben Eltern-Knoten P haben, es also sowohl eine Kante E von P nach C1 als auch eine Kante F von P nach C2 gibt.

- **Vorfahr-Knoten**
  Ein Knoten A ist ein Vorfahre von Knoten D, wenn es einen Pfad von A nach D gibt.

- **Nachkommen-Knoten**
  Ein Knoten D ist ein Nachkomme von Knoten A, wenn es einen Pfad von A nach D gibt.

- **Wurzel-Knoten**
  Ein Wurzel-Knoten ist ein Knoten welcher keinen Eltern-Knoten hat (in HTML Dokumenten ist dies der Knoten, den das ***html*** Element bildet).

- **Blatt-Knoten**
  Ein Blatt-Knoten ist ein Knoten welcher keine Kind-Knoten hat.

[3] Die Beziehung der DOM-Knoten untereinander ist von großer Relevanz für das nun folgende Kapitel.

2.1.3 CSS

CSS steht für **C**ascading **S**tyle **S**heets und beschreibt, wie HTML-Elemente in verschiedenen Medien wie auf Bildschirmen oder in einem Ausdruck dargestellt werden sollen. Während HTML dafür benutzt wird, den Inhalt einer Webseite selbst zu beschreiben (vgl. Kapitel 2.1.1), dient CSS also der Beschreibung der inhaltlichen Darstellung.

Vor der Einführung von CSS gab es eine starke Vermischung von Inhalt und Darstellung. Bei der Definition von HTML Tags gab es mit Einführung der Version 3.2 die Möglichkeit, einem HTML Tag spezielle Attribute zuzuweisen, welche die Darstellung bestimmt haben (z.B. ***bgcolor*** und ***border***). Außerdem wurden eigene HTML Elemente wie ***font***, ***u*** und ***center*** eingeführt, die ausschließlich die Darstellung von weiteren HTML-Elementen, die als Kind dieser Elemente eingebunden waren, beeinflussen sollten.

Seit HTML5 sind viele der oben genannten Attribute und HTML Tags als veraltet markiert worden und es wird empfohlen, ihre Funktionalität mit CSS abzubilden. [4]

---

[3] (van Kesteren, et al. 2015)
[4] (Hickson, et al. 2014)

Die in dieser Arbeit beschriebenen Begriffe und Beispiele stützen sich auf Features und empfohlene Vorgehensweisen, welche sich mit CSS3 etabliert haben.

2.1.3.1 **Regeln**

Eine CSS-Datei besteht aus einer Menge von CSS-Regeln. Es gibt zwei verschiedene Arten von Regeln:

- **at-Regeln**
  Der Name dieses Typs leitet sich daraus ab, dass ein „@“ Zeichen ihn einleitet, welchem ein Identifier folgt. Er wird unter anderem dazu verwendet, andere CSS-Dateien zu importieren, neue Schriftarten zu definieren oder sogenannte Media-Queries abzubilden.
  Da diese Bearbeitung sich nicht weiter mit den at-Regeln befasst, werden sie hier nur der Vollständigkeit halber aufgeführt und nicht näher erläutert.

- **Regelsätze**
  Diese Art wird manchmal der Einfachheit nur als Regel bezeichnet, da sie am häufigsten verwendet wird. Ein Regelsatz besteht aus zwei Teilen:

  - Zunächst folgen ein oder mehrere durch Komma getrennte CSS-Selektoren
  - Danach folgt innerhalb einer geschweiften Klammer der Regelblock

[5]

2.1.3.2 **Selektoren**

Der komplexeste Teil einer CSS Regel sind die Selektoren, welche die DOM-Knoten identifizieren sollen, für die die angegebenen Deklarationen angewandt werden sollen.

Wenn mehrere Selektoren die gleichen Deklarationen teilen, können sie als komma-separierte Liste kombiniert werden. Das Komma agiert hierbei als logisches ODER, also wird die Deklaration angewandt, wenn einer der Selektoren zutrifft.

Ein Selektor wiederum besteht aus einem oder mehreren sogenannten einfachen Selektoren, die nach bestimmten Regeln kombiniert werden können.

---

[5] (Lie und Bos 1996)

Folgende Selektoren bezeichnet man als einfache Selektoren:

- **Typ-Selektor**
  Sucht nach DOM-Knoten eines bestimmten Element-Typs. In der Regel entspricht dieses dem verwendeten HTML Tag Namen.

```
a, div, p, input {}
```

- **Universal-Selektor**
  Dieser Selektor entspricht allen DOM-Knoten.

```
* {}
```

- **Klassen-Selektor**
  Sucht nach DOM-Knoten, welche eine bestimmte CSS Klasse haben

```
.text, .button, .my-class {}
```

- **ID-Selektor**
  Analog zum Klassen-Selektor wird hier nach DOM-Knoten mit einer bestimmten ID gesucht.

```
#text-id, #primarybutton {}
```

- **Attribut-Selektor**
  Sucht nach DOM-Knoten, die ein bestimmtes (gesetztes) Attribut haben. Optional kann auch der Wert des Attributes bei der Selektion eine Rolle spielen.

```
[placeholder], [data-id="123"], [data-name~="a"] {}
```

- **Pseudo-Element-Selektor**
  Selektiert innerhalb eines DOM-Knoten spezielle als Pseudo-Element bezeichnete Rendering Abschnitte.

```
::first-line, ::first-letter, ::before, ::after, ::selection {}
```

- **Pseudo-Klassen-Selektor:**
  Selektiert DOM-Knoten welche entweder bestimmte strukturellen oder dynamischen Pseudoklassen zugeordnet werden können oder mit Hilfe von den Bedingungen Selektoren innerhalb von ***lang***, ***not*** und ***matches*** genügen.

Letztere seien nur der Vollständigkeit halber erwähnt und werden nicht weiter betrachtet.

```
:first-child, :nth-child(2+n), :hover, :focus, :not(a), :not(:hover) {}
```

Die oben genannten einfachen Selektoren lassen sich mit Ausnahme des Element- und des Universal-Selektors auf beliebige Art kombinieren, um anzugeben, dass die zu findenden DOM-Knoten mehrere Eigenschaften erfüllen müssen. Hierbei werden die Teil-Selektoren hintereinander ohne Leerzeichen angegeben. Der Element- und der Universal-Selektor schließen sich gegenseitig aus und dürfen nur einmal zu Beginn des Teil-Selektors auftreten.

Die folgende Abbildung zeigt einige Beispiele für komplexere Selektoren.

```
a {}
div.my-class {}
div#my-id {}
#my-id {}
* {}
*.my-class:hover::before {}
.my-class:hover::before {}
```

Abbildung 1: Beispiele für die Kombination von einfachen Selektoren

Der Universal-Selektor findet (im Gegensatz zum Beispiel) in der Praxis normalerweise nur dann Verwendung, wenn ein DOM-Knoten keine bestimmte Eigenschaft erfüllen muss, um bei einer Selektion in Betracht gezogen zu werden.

Im obigen Beispiel wählen die letzten beiden Selektoren in jedem Fall die gleichen DOM-Knoten aus.

Bei der Suche nach einem DOM-Knoten können auch vorausgegangene DOM-Knoten in Betracht gezogen werden. Hierfür gibt es ebenfalls spezielle Selektoren, sogenannte Kombinator-Selektoren:

- **Kind-Selektor „>“**
  Wählt DOM-Knoten aus, die den rechten Selektor erfüllen und ein Kind eines DOM-Knotens sind, welcher den linken Selektor erfüllt

```
div > .my-class, :hover > :first-child {}
```

- **Nachfahren-Selektor „ „**
  Wählt DOM-Knoten aus, die den rechten Selektor erfüllen und ein Nachfahre eines DOM-Knotens sind, welcher den linken Selektor erfüllt

```
article p, #main .content, .list * {}
```

- **Nachbar-Selektor**
  Wählt DOM-Knoten aus, die den rechten Selektor erfüllen und deren linker Nachbar den linken Selektor erfüllt

```
h2 + h2, .title + p {}
```

- **Geschwister-Selektor**
  Wählt DOM-Knoten aus, die den rechten Selektor erfüllen und mindestens einen Geschwister-Knoten haben, der vor ihm definiert wurde, der den linken Selektor erfüllt

```
div ~ div, input:checked ~ input {}
```

Auch Kombinator-Selektoren lassen sich wieder kombinieren, wodurch es möglich ist sehr spezifische und komplexe Regeln zu definieren. [6]

2.1.3.3 **Deklarationen**

Wie im vorherigen Kapitel beschrieben, findet eine Deklaration Anwendung, wenn einer oder mehrere Selektoren des Regelsatzes, in dem sich die Deklaration befindet, auf ein DOM Element zutreffen. Der zutreffende DOM-Knoten ist immer der letzte DOM-Knoten, der bei der Auswertung des Selektors von links nach rechts in Betracht gezogen wird.

Deklarationen sind Key-Value Paare, welche durch einen Doppelpunkt voneinander getrennt sind. Jede Deklaration wird mit einem Semikolon abgeschlossen. Als besonders wichtig geltende Deklarationen können mit dem Schlüsselwort ***!important*** versehen werden, um ihre Relevanz bei der Bestimmung, welche Deklaration für einen DOM-Knoten zutrifft, zu erhöhen (mehr dazu im nächsten Kapitel).

In anderen CSS-Regeln definierte Deklarationen lassen sich nicht zurücksetzen, es muss also immer ein Wert angegeben werden. Zwar existiert ein Wert: ***inherit***, dieser beschreibt allerdings nicht die Verwendung des Werts des letzten zutreffenden Regelsatz sondern die Vererbung des Werts, den der Elternknoten innerhalb des DOMs hat, was ein signifikanter Unterschied ist. [7]

---

[6] (Çelik, et al. 2011)
[7] (Lie und Bos 1996)

2.1.3.4 **Kaskade**

Die Kaskade oder Cascade, die Teil des Namens „Cascading Style Sheets“ ist, bestimmt, welche Deklaration für ein DOM-Element angewendet wird, wenn mehrere Regeln zutreffen und es verschiedene mögliche Werte für eine Eigenschaft gibt.

Der vollständige, in der W3C CSS Definition beschriebene Algorithmus sieht folgendermaßen aus:

1. Find all declarations that apply to the element and property in question, for the target media type. Declarations apply if the associated selector matches the element in question and the target medium matches the media list on all @media rules containing the declaration and on all links on the path through which the style sheet was reached.
2. Sort according to importance (normal or important) and origin (author, user, or user agent). In ascending order of precedence:
    1. user agent declarations
    2. user normal declarations
    3. author normal declarations
    4. author important declarations
    5. user important declarations
3. Sort rules with the same importance and origin by specificity of selector: more specific selectors will override more general ones. Pseudo-elements and pseudo-classes are counted as normal elements and classes, respectively.
4. Finally, sort by order specified: if two declarations have the same weight, origin and specificity, the latter specified wins. Declarations in imported style sheets are considered to be before any declarations in the style sheet itself.

Abbildung 2: Cascading Algorithmus (Bos, et al. 2016)

Diese Bearbeitung befasst sich nicht weiter mit Media-Queries, darum gilt vereinfacht folgendes:

1. Sammle alle Deklarationen aus allen CSS-Dokumenten ein, die auf den aktuellen DOM-Knoten und die festzulegende Eigenschaft zutreffen.
2. Sortiere die Werte nach ihrer Wichtigkeit (normal und important) und ihrer Herkunft in aufsteigender Reihenfolge nach folgender Vorrangigkeit:
    1. **Deklarationen aus dem User Agent**
       Dies ist in den meisten Fällen der Webbrowser. Dieser hat häufig Standard-Werte für bestimmte Properties in Abhängigkeit zum Tag gesetzt

    2. **Normale Deklarationen des Benutzers**
       Ein Benutzer kann im Webbrowser ein persönliches Stylesheet verwenden, welches nicht zur eigentlichen Webseite gehört. Mit normalen Deklarationen sind diejenigen Deklarationen gemeint, die nach ihrem Wert nicht das Schlüsselwort ***!important*** benutzen.

3. **Normale Deklarationen des Autors (der Webseite)**
   Hierzu zählen nicht-***!important*** Deklarationen aus allen CSS-Dateien, auf die die Webseite verweist.

4. **Important Deklarationen des Autors (der Webseite)**
   Analog dazu sind dies alle ***!important*** Deklarationen auf die die Webseite verweist.

5. **Important Deklarationen des Benutzers**
   Die höchste Vorrangigkeit haben die als ***!important*** definierten Deklarationen des Benutzers.

3. Innerhalb aller Deklarationen mit gleicher Wichtigkeit und Herkunft wird nun nach der Spezifität der Selektoren innerhalb der Regelmenge sortiert. Hierbei werden die verwendeten Selektoren einer bestimmten Gruppe gezählt.

   Heraus resultieren folgende Variablenbelegungen

   1. ***d*** = Anzahl der Typ-Selektoren und Pseudo-Element-Selektoren ()
   2. ***c*** = Anzahl der Klassen-Selektoren und Pseudo-Klassen-Selektoren
   3. ***b*** = Anzahl der ID-Selektoren

   Es existiert eine weitere Belegung:

   4. ***a*** = Wenn die Deklaration aus dem ***style*** Attribut des DOM-Knotens kommt (welches nur einmal auftreten darf) 1, sonst 0

   Würde man nun ***a***, ***b***, ***c*** und ***d*** in einem Stellenwert-System betrachten dessen Basis ***basis*** größer ist als der maximale Wert von a, b, c und d in allen Selektoren berechnet wird, so würde zur Berechnung der Spezifität gelten:

$$specificy = a * basis^3 + b * basis^2 + c * basis^1 + d * basis^0$$
$$\leftrightarrow specifiy = a * basis^3 + b * basis^2 + c * basis + d$$

   Die Deklarationen gleicher Wichtigkeit und Herkunft werden aufsteigend nach der errechneten Spezifität sortiert.

4. Innerhalb aller Deklarationen gleicher Wichtigkeit, Herkunft und Spezifität wird am Ende nach der Reihenfolge der Deklaration sortiert.

Die Reihenfolge bestimmt sich nach dem Auftreten der jeweiligen Deklaration, wenn der DOM ausgehend vom Wurzel-Knoten mit einer Tiefensuche durchlaufen wird (was in Regel bereits beim ersten Auslesen stattgefunden hat).

Innerhalb von CSS-Dateien werden Deklarationen aus anderen CSS-Dateien, die mit Hilfe der at-Regel ***@import*** eingebunden wurden, unabhängig von der wirklichen Position der at-Regel in der einbindenden Datei, so behandelt, als wären die Deklarationen vor allen eigenen Deklarationen der CSS-Datei definiert worden.

Angewandt wird nach dieser Berechnung die Deklaration, die am Ende der Liste steht. [8]

2.2 Objektorientierte Programmierung

Das zentrale Konstrukt der objektorientierten Programmierung ist das Objekt. Dieses repräsentiert eine „Zusammenfassung von Daten (Zuständen des Objekts) und der dazugehörenden Funktionalität (den vom Objekt unterstützten Operationen)“.

Ein Objekt schützt seinen inneren Zustand vor direktem Zugriff (vgl. Kapitel 2.2.1.3). Dies erfolgt mit Hilfe von Operationen auf Objekten. Die Sammlung eben dieser bezeichnet man als Schnittstelle eines Objekts. [9]

Da die Definition der einzelnen Aspekte der objektorientierten Programmierung sehr generisch ist, wird das folgende Kapitel sich auf die Aspekte beschränken, die sich in Java wiederfinden und sie darüber beschreiben.

2.2.1 Java

Java ist eine objektorientierte Programmiersprache der Firma Sun. Die in dieser Sprache geschriebenen Programme sind – solange es für das jeweilige System eine Java-Laufzeitumgebung gibt – plattformunabhängig, d.h. sie können ohne weitere Änderungen auf jeder Rechnerarchitektur ausgeführt werden. [10]

2.2.1.1 **Klassen**

Objekte lassen sich klassifizieren also einer oder mehreren Klassen zuordnen. Eine Klasse definiert die Gemeinsamkeiten der ihr zugeordneten Objekte. [11]

---

[8] (Bos, et al. 2016)
[9] (Lahres und Rayman 2009)
[10] (Lemay und Cadenhead 2000)
[11] (Lahres und Rayman 2009)

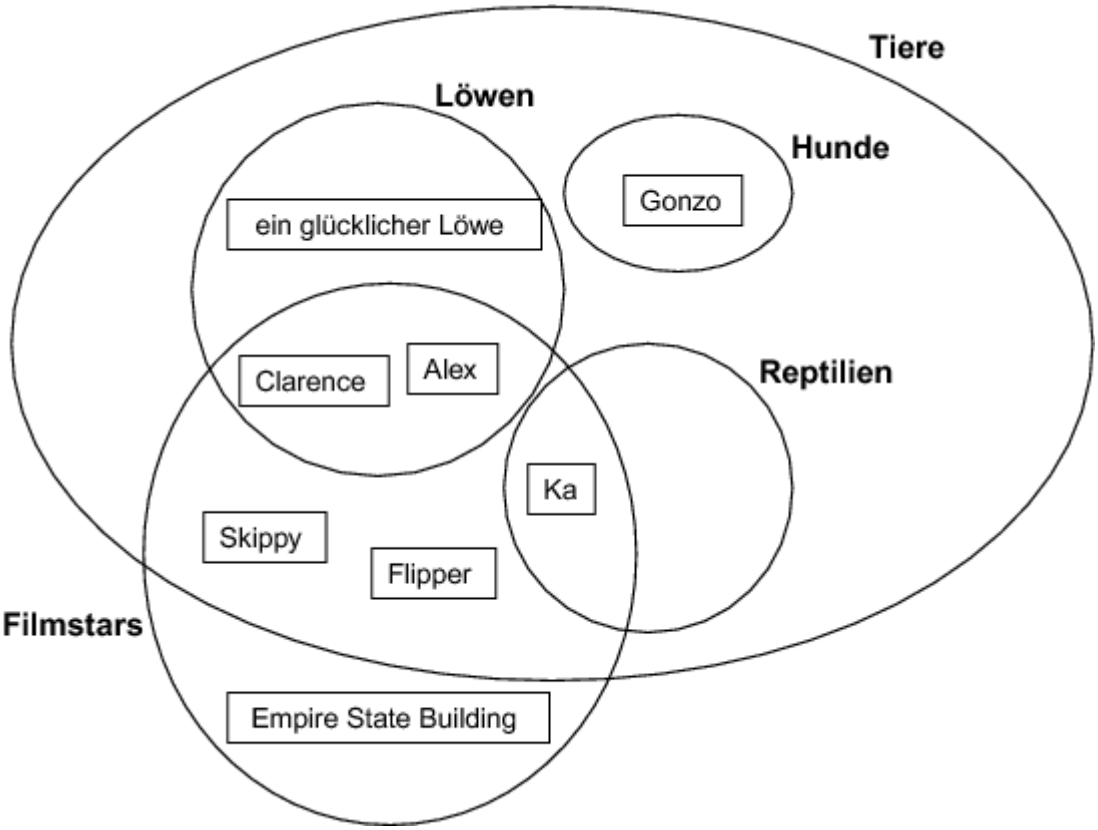


Abbildung 3: (Lahres und Rayman 2009)

Abbildung 3 zeigt ein Beispiel für Objekte und Klassen. Hierbei repräsentieren die Ellipsen mit ihren jeweils fettgedruckten Namen die Klassen und die von Rechtecken umgebenen Namen die jeweiligen Objekte. So gehört das Objekt ***Flipper*** zum Beispiel der Klasse ***Filmstars*** und der Klasse ***Tiere*** an. [12] In Java ist eine Klasse der Bauplan für ein Objekt, welches man dann hinsichtlich seiner Beziehung zu der Klasse als Instanz dieser Klasse bezeichnet. Die Klasse setzt sich aus Attributen und Methoden zusammen, die in den nachfolgenden Kapiteln beschrieben werden. [13]

2.2.1.2 **Attribute**

Als Attribut bezeichnet man allgemein Eigenschaften, die das Objekt beschreiben und Daten, die das Objekt verwaltet. In Java wird dies über Variablen, die an der Klasse definiert werden, abgebildet. Eine Variablendefinition besteht aus einem in ihrem Gültigkeitsbereich (wird hier nicht weiter erläutert) noch nicht (neu) definierten Namen und einem Typ.

---

[12] (Lahres und Rayman 2009)
[13] (Lemay und Cadenhead 2000)

Der Typ der einer Variable ist in der Regel ein nativer Datentyp (z.B. Ganzzahl-Werte, Fließkommazahlen, …) oder eine andere Klasse.

Es wird zudem zwischen Klassenvariablen und Instanzvariablen unterschieden. Während eine Klassenvariable den gleichen Wert für alle Instanzen der Klasse behält, also direkt in der Klasse gespeichert wird, hält eine Instanzvariable für jede Instanz einen eigenen Wert. [14]

2.2.1.3 **Methoden**

In Java versteht man unter einer Methode das Verhalten einer Klasse, welche man über einen bestimmten Bezeichner – den Namen der Methode - aufrufen kann. Methoden können zum Beispiel andere Methoden und Variablen innerhalb ihres Sichtbarkeitsbereichs aufrufen bzw. lesen und schreiben (vgl. Kapitel 2.2.1.7).

Bei ihrer Definition muss für eine Methode eine Parameterliste angeben, auf sie dann während ihrer Ausführung Zugriff hat und welche ihr Verhalten beeinflussen können.

Die Kombination aus Name und Parameterliste einer Methode muss innerhalb einer Klasse eindeutig sein.

Auch für Methoden gibt es analog zu Attributen eine Unterscheidung zwischen Klassenmethoden und Instanzmethoden. [15]

2.2.1.4 **Interfaces**

Das Verhalten einer Operation innerhalb eines Objekts wird mit Hilfe eines Kontrakts geregelt. Dieser definiert die Rahmenbedingungen, wozu die Vor- und Nachbedingungen für den Aufruf einer Operation sowie ggf. Invarianten - also Bedingungen, die immer gelten sollen - zählen. [16]

In Java gibt es hierfür unter anderem das sogenannte Interface, welches von Klassen implementiert werden kann. Eine Klasse kann eine beliebige Menge von Interfaces implementieren, indem es alle seine in der Definition beschrieben Methoden implementiert. Indem sie ein Interface implementiert wird angenommen, dass der Kontrakt, den das Interface definiert hat, von der Klasse eingehalten wird.

2.2.1.5 **Kapselung von Daten**

---

[14] (Lemay und Cadenhead 2000)
[15] (Lemay und Cadenhead 2000)
[16] (Lahres und Rayman 2009)

Unter der Kapselung versteht man das Verbergen von Implementierungsdetails indem auf die interne Datenstruktur eines Objekts nicht direkt zugegriffen werden darf. Stattdessen werden entsprechende Schnittstellen in Form von Methoden definiert mit deren Hilfe der Zugriff auf die Daten eines Objekts ermöglicht werden kann. Hierdurch ist es möglich die interne Datenstruktur einer Klasse zu verändern ohne dass Änderungen an anderen Stellen nötig sind. [17] Außerdem bestimmt das Objekt welche Daten überhaupt les- oder schreibbar sind und kann die Konsistenz eben dieser sicherstellen. Dies ist zum Beispiel erforderlich, wenn mehrere Werte voneinander abhängig sind und eine Änderung den jeweiligen anderen Wert beeinflusst. [18]

2.2.1.6 **Polymorphie**

Die Polymorphie bezieht sich auf das unterschiedliche Verhalten verschiedener Objekte bei dem Aufruf einer Funktion. [19] In Java bedeutet dies, dass man eine Methode gleichen Namens mehrfach definiert werden kann, solang sie an Hand ihrer Parameterliste unterschieden werden kann. Dies kann dann zu einem anderen Verhalten der Methode führen. [20]

2.2.1.7 **Vererbung und Zugriffskontrolle**

Unter Vererbung versteht man, dass eine neue Klasse die Methoden und Attribute von einer anderen Klasse erbt. Die so entstandene Klasse bezeichnet man in Relation zu der Klasse, von der sie erbt, als Unterklasse oder abgeleitete Klasse.

Dies heißt aber nicht, dass die Klasse auf alle Methoden und Attribute zugreifen kann. Die Sichtbarkeit bestimmt in Java das Prinzip der Zugriffskontrolle. Es gibt 4 Schutzebenen, die man für jede einzelne Methoden und Attribute einer Klasse festlegen kann:

- **Privat (Schlüsselwort: private)**
  Die so definierten Methoden und Attribute sind nur innerhalb der eigenen Klasse sichtbar. Erbt eine neue Klasse von dieser Klasse, hat diese keinen Zugriff darauf.

- **Geschützt (Schlüsselwort :protected)**
  wie bei privat, aber auch für Unterklassen sichtbar

- **Paket (kein explizites Schlüsselwort)**

---

[17] (Lahres und Rayman 2009)
[18] (Lahres und Rayman 2009)
[19] (Lahres und Rayman 2009)
[20] (Lemay und Cadenhead 2000)

Für so definierte Methoden und Attribute bedeutet dies, dass sie von allen Klassen in dem gleichen Package verwendet werden können. Dies gilt, wenn kein Schlüsselwort für die Sichtbarkeit angegeben wurde.

- **Öffentlich (Schlüsselwort: public)**
  Diese Methoden und Attribute sind unbeschränkt sichtbar.

Während andere Programmiersprachen das Prinzip der Mehrfachvererbung unterstützen, ist es einer Klasse in Java lediglich möglich von einer Klasse zu erben. [21]

2.3 Entwurfsmuster

Bei vielen Problemstellungen wird das Rad nicht mit jeder Lösung neu erfunden. Vielmehr werden die Kernideen, die andere bereits bei ähnlichen Problemstellungen herausgearbeitet haben, aufgegriffen und in die neue Lösung integriert.

Ein Entwurfsmuster ist ein bewährtes Rezept für eine Herangehensweise an ein immer wiederkehrendes Problem. [22]

Es werden vier maßgebliche Elementen definiert, die ein Entwurfsmuster auszeichnen:

1) Der Name des Patterns als Referenz zum jeweiligen Designproblem.
2) Die Problemstellung unter der das Pattern angewendet werden kann. Hier werden auch Bedingungen angegeben, unter denen die Nutzung des Patterns sinnvoll ist.
3) Die Lösung, die das Pattern vorsieht, welche aber keine konkrete Implementierung vorsieht, sondern lediglich die Beziehung der Design bildenden Elemente verdeutlichen soll.
4) Konsequenzen, die sich bei der Nutzung des Patterns ergeben.

[23]

2.3.1 B.E.M. Pattern

Das „Block Element Modifier“ – Pattern ist ein Benennungsmuster (vgl. Kapitel 2.3), welches für CSS Klassen verwendet wird, um Ihnen allein durch ihre Benennung mehr Bedeutung und Struktur zukommen zu lassen. Es eignet sich besonders für große Projekte, die über längere Zeit hinweg Bestand haben sollen.

Initial wurde das Pattern durch Programmierer von Yandex, einem Unternehmen für Internetdienstleistungen, angewandt und bekannt gemacht. Mittlerweile gibt es eine Reihe

---

[21] (Lemay und Cadenhead 2000)
[22] (Buschmann, et al. 1996)
[23] (Gamma, et al. 2015)

von Abwandlungen und Modifikationen des Musters. Diese Arbeit wird sich auf eine Modifikation von Nicolas Gallagher stützen.

Wie der Name des Musters bereits schließen lässt, besteht es aus drei wichtigen Teilen:

- Bei einem Block handelt es sich um eine höhere Abstraktionsebene, z.B. eine Komponente
- Dieser wiederum kann aus einem oder mehreren Elementen bestehen, die ihn definieren
- Eine Komponente kann verschiedene Zustände oder Ausprägungen haben, welche als Modifier bezeichnet werden

Diese drei Teile werden durch CSS Klassennamen (vgl. Kapitel 2.1.3.2) repräsentiert, die DOM-Knoten (vgl. Kapitel 2.1.2) zugewiesen werden. Sie werden nach folgendem Namens-Schema erzeugt:

- Für den Block dürfen alle nach CSS Syntax gültigen Zeichenfolgen als Klassennamen verwendet werden, er darf allerdings nicht zwei aufeinander folgende Unterstriche (__) oder zwei aufeinander folgende Bindestriche (--) beinhalten, da diese für Element- bzw. Modifier-Klassennamen verwendet werden.

```
article
video-player
```

- Für Elemente wird der Name des Blocks (für den dann auch die oben definierten Regeln gelten), gefolgt von zwei aufeinander folgenden Unterstrichen (__), gefolgt von einem eindeutigen Elementnamen als CSS Klassenname verwendet.

```
article__title
video-player__play-button
```

- Analog dazu besteht ein Modifier Klassenname aus dem Namen des Blocks, gefolgt von zwei aufeinander folgenden Bindestrichen (--), gefolgt von einem eindeutigen Modifiernamen.

```
article--highlighted
video-player--playing
```

Innerhalb des DOMs stehen DOM-Knoten, die B.E.M. Klassennamen verwenden in folgender Relation zueinander:

1) Element Klassennamen dürfen nur in Nachfahren des DOM-Knotens verwendet werden, welche den jeweiligen Block Klassennamen trägt.

2) Modifier Klassennamen dürfen nur innerhalb des DOM-Knotens verwendet werden, welches den Block Klassennamen enthält. Hieraus folgt, dass ein Modifier Klassenname auch niemals ohne einen Block Klassennamen verwendet wird.

Ansonsten gibt es nach Definition keine Einschränkungen bei der Verwendung innerhalb eines DOMs. Alle Blöcke, Elemente und Modifiers können im DOM mehrfach auftreten. Innerhalb eines Blocks darf auch dieselbe Elementklasse an mehreren DOM-Elementen benutzt werden. [24]

```
<article class="article">
  <h2 class="article__title">B.E.M. Beispiel 2</h2>
  <div class="article__content">
    <p class="text">Das folgende Video dient als weiteres Beispiel</p>
    <div class="video-player">
      <button class="video-player__play-button">Abspielen</button>
      <video class="video-player__video" src="video.mp4"></video>
    </div>
  </div>
</article>
```

Abbildung 4: Beispiel für die Verwendung von B.E.M. in HTML

2.4 IntelliJ IDEA

2.4.1 Allgemein

IntelliJ IDEA ist eine Entwicklungsumgebung (IDE) von der Firma ***JetBrains***, welche im Jahr 2001 zum ersten Mal erschienen ist. Neben IntelliJ IDEA gibt es eine Reihe von weiteren IDEs, welche alle mit einem gemeinsamen Kern laufen, dem ***IntelliJ SDK***. [25]

2015 hatte das Unternehmen 400.000 Kunden aus 180 Ländern. Die 580 Mitarbeiter teilen sich auf die 5 Standorte Prag, St. Petersburg, Boston, Moskau und München auf. Die einzelnen Produkte sollen Entwickler bei der täglichen Arbeit unterstützen, indem sie eine Vielzahl von Möglichkeiten zur Code-Analyse, Korrektur und Anpassung (Refactoring) bieten. [26]

[24] (Friedman, et al. 2013)
[25] (JetBrains Documentation Team 2016)
[26] (JetBrains Homepage Team 2016)

2.4.2 Begriffe

Die folgenden Kapitel beschreiben Begriffe, die für die Plugin-Entwicklung in IDEA relevant sind.

2.4.2.1 **Virtuelle Datei**

Eine virtuelle Datei ist die Darstellung, die in der IntelliJ Plattform genutzt wird, um eine Datei aus einem Dateisystem darzustellen. Meistens ist dies eine Datei auf der Festplatte, allerdings abstrahiert das virtuelle Dateisystem auch andere Quellen, wie zum Beispiel ein JAR-Archiv oder eine alte Version einer Datei, die aus einem Versionskontrollsystem kommen kann. Dies hat den Vorteil, dass die anderen Schichten sich nicht mit Besonderheiten, wie zum Beispiel Kodierungen befassen müssen, sondern eine virtuelle Datei einfach als ein Datenstrom aus Bytes behandelt werden kann. [27]

2.4.2.2 **Dokument**

In einem Dokument befindet sich eine editierbare Folge von Unicode Zeichen, deren Inhalt in der Regel aus einer virtuellen Datei stammt. Zeilenumbrüche werden unabhängig vom Betriebssystem immer mit einem ***\n*** eingeleitet. Die Umwandlung erfolgt erst, wenn das Dokument geladen oder gespeichert wird.

Das Dokument ist das, was der Benutzer mit Hilfe eines Editors direkt mit Tastatureingaben editiert, aber auch die normalisierte Darstellung, die von höheren Schichten zu Analyse weiterverarbeitet wird. [28]

2.4.2.3 **AST**

Der „Abstract Syntax Tree“ oder kurz AST ist der erste Verarbeitungsschritt, wenn der Inhalt eines Dokuments strukturiert dargestellt werden soll. Die einzelnen AST-Knoten, aus denen der gewurzelte Baum besteht, bilden ab, wie die einzelnen Token, die ein Lexer erzeugt hat, zusammenhängen. Die Blatt-Knoten des Baums bilden hierbei direkt die Tokens eines bestimmten Typs ab, während der Wurzel-Knoten die Datei darstellt.

Jeder AST-Knoten erhält eine Abbildung auf den Text-Bereich, den er innerhalb der Datei repräsentiert. Änderungen, die am AST-Baum vorgenommen werden, werden direkt in das Dokument übertragen. [29]

---

[27] (JetBrains Documentation Team 2016)
[28] (JetBrains Documentation Team 2016)
[29] (JetBrains Documentation Team 2016)

2.4.2.4 **PSI**

PSI steht für „Program Structure Interface“ und stellt eine strukturierte Repräsentation des Inhalts einer Datei in einer bestimmten (Programmier-)Sprache dar, der mit dem AST-Baum des Dokuments verknüpft ist. Meistens werden die dafür in Java erforderliche Klassen in IntelliJ Plugins mit Hilfe eines Parsers, der auf den vom Lexer generierten Tokens arbeitet, generiert.
Die einzelnen Teile des PSI-Baums werden als PSI-Element bezeichnet und sind ebenfalls in einem gewurzelten Baum angeordnet. Ein besonderes PSI-Element ist die PSI-Datei, welche den Wurzel-Knoten des Baums darstellt.

Der Hauptunterschied zwischen dem AST und PSI ist, dass ein AST-Knoten im Gegensatz zu PSI-Elementen keine sprachspezifischen Semantiken und Methoden enthält. [30] Somit gilt eine ähnliche Trennung wie für Lexer und Parser (vgl. (Appel und Palsberg 2012)).

2.4.2.5 **Sprachinjektion**

Um eine neue Sprache in eine bestehende Sprache einzubetten, gibt es in IDEA ein Feature mit dem Namen Sprachinjektion („Language Injection“). Hierbei wird ein PSI-Element – nachfolgend „Language Injection Host“ genannt - einer bestehenden Sprache bestimmt, dessen String-Repräsentation in einer anderen Sprache – nachfolgend injizierte Sprache - geparst werden soll.

Voraussetzung für die Verwendung von Sprachinjektion ist das ***IntelliLang*** Plugin, welches mit IDEA ausgeliefert wird und aktiviert sein muss.

Das Plugin bietet eine Benutzungsoberfläche mit deren Hilfe man Sprachinjektion mit Hilfe von Menüs und Formularen vornehmen kann. Die Integration beschränkt sich allerdings auf ausgewählte String Literale (zum Beispiel XML Attribut Werte).

Da es für die injizierte Sprache nicht immer nötig sein muss, einen vollständigen PSI-Baum innerhalb des „Language Injection Host“s zu definieren, ist es möglich, einen Präfix und einen Suffix für die Injektion zu definieren, welche dann zusammen mit dem Inhalt des Hosts von dem Parser geparst werden. [31]

---

[30] (JetBrains Documentation Team 2016)
[31] (JetBrains Documentation Team 2016)

2.4.3 Plugin Entwicklung

Die Plugin-Entwicklung erfolgt in der IDE selbst durch die Erstellung eines neuen ***IntelliJ Platform Plugin***-Projekts. Hierbei bildet die dabei automatisch erstellte ***plugin.xml*** den Kern des Plugins, indem sie unter anderem folgende Informationen bereitstellt:

- **<id>**
  Die eindeutige ID des Plugins

- **<name>**
  Der Name des Plugins

- **<version>**
  Die Version des Plugins

- **Weitere Metainformationen**
  z.B. Autor, Beschreibung, Change-Notes

- **<depends>**
  Abhängigkeiten zu anderen Plugins oder Packages

- **<resource-bundle>**
  Ein Resource-Bundle, in dem sich z.B. Lokalisierungen für Tooltips befinden können

- **<extensions>**
  Eine Liste der verwendeten Erweiterungspunkte und ihrer Konfiguration (werden im nächsten Kapitel behandelt)

- **<extensionPoints>**
  Eine Liste der vom Plugin zur Verfügung gestellten Erweiterungspunkte (wird im Folgenden nicht weiter beschrieben)

[32]

2.4.4 Erweiterungspunkte

Ein Weg, Funktionalitäten in einem Plugin bereitzustellen, geht über die Verwendung von Erweiterungspunkten.

Neben der Code-Dokumentation in Form von JavaDoc an auswählten Java-Klassen [33] stellt JetBrains zwei Kapitel „Custom Language Support“ und „Custom Language Support

[32] (JetBrains Documentation Team 2016)

Tutorial“ zur Verfügung, welche die Verwendung einiger Erweiterungspunkte zeigt und mit Beispielen unterlegt. [34] Die folgenden Kapitel zeigen die für die geplanten Funktionen des B.E.M. Plugins relevanten Erweiterungspunkte.

Als jeweilige Überschrift wird hierbei der Name des Erweiterungspunkts genommen, wobei auf die Angabe von ***com.intellij.*** verzichtet wird, da sich kein Erweiterungspunkt außerhalb dieses Packages befindet.

2.4.4.1 Erweiterungspunkt: fileTypeFactory

Bei der Entwicklung eines Sprach-Plugins ist es möglich einen Dateitypen zu registrieren, mit dem die Sprache assoziiert wird. Hierbei wird die Basis-Klasse ***FileTypeFactory*** abgeleitet und die Methode ***createFileTypes*** implementiert, welche einen Parameter ***consumer*** vom Typ ***FileTypeConsumer*** erhält. An diesen Parameter wird die Methode ***consume*** aufgerufen, welche zwei Parameter erwartet:

- ***fileType*** vom Typ ***FileType***:
  Legt eine Implementierung vom Typ ***FileType*** fest, welche den Dateitypen spezifiziert. Es wird vorgeschlagen, die Klasse ***LanguageFileType*** zu erweitern

- ***extensions*** vom Typ ***String*** oder ***matchers*** vom Typ ***FileNameMatches***:
  Bestimmt, welche Dateinamen akzeptiert werden.

Das Interface ***FileType*** legt unter anderem die folgenden Eigenschaften fest:

- Language
- Name
- Beschreibung
- Standard Dateinamen-Erweiterung
- Icon

Während alle übrigen Eigenschaften für die Benutzungsoberfläche relevant sind, stellt die erste Eigenschaft die Implementierung der Sprache dar. Die abzuleitende Klasse ***Language*** erwartet hierbei lediglich die Implementierung eines Konstruktors.

Die für das B.E.M. Plugin relevante Konstruktor-Variante erwartet lediglich eine ID in Form einer Zeichenkette. Diese ist für viele weitere sprachabhängige Erweiterungspunkte relevant, um die Sprache eindeutig zu identifizieren.

---

[33] (JetBrains Development Team 2016)
[34] (JetBrains Documentation Team 2016)

2.4.4.2 Erweiterungspunkt: lang.parserDefinition

Zur Parser Definition gehört neben der Erzeugung des Parsers für ein Projekt auch die Erzeugung eines Lexers, die Angabe eines ***FileNodeTypes*** für die Sprache, einer Fabrik Methode zur Erzeugung von PSI-Elementen aus AST Nodes sowie einer Menge von Methoden, die die erzeugten Token des Lexers klassifizieren oder in Beziehung zueinander setzen.

2.4.4.3 Erweiterungspunkt: syntaxHighlighterFactory

Mit Hilfe dieses Erweiterungspunkts kann ein Syntax-Highlighter für eine Sprache definiert werden. Hierzu muss die eindeutige ID der Sprache übergeben werden und eine Implementation des Interfaces ***SyntaxHighlighterFactory***.

Die Implementation muss die Methode ***getSyntaxHighlighter*** bereitstellen, welche als Parameter das aktuelle Projekt und die virtuelle Datei erhält, für die der Syntax-Highlighter erstellt werden soll. [35]

2.4.4.4 Erweiterungspunkt: colorSettingsPage

Der Erweiterungspunkt registriert eine Seite mit einem Formular, über das Farbeinstellungen für das Syntax-Highlighting einer Sprache festgelegt werden können. Hierfür muss das Interface ***ColorSettingsPage*** über folgende Methoden implementiert werden:

- **getIcon**
  Liefert das Icon für die Seite

- **getHighlighter**
  Liefert die Implementation eines ***SyntaxHighlighter***s, der für die Vorschau-Ansicht auf der Seite benötigt wird

- **getDemoText**
  Liefert den Text zurück, der in der Vorschau-Ansicht angezeigt werden soll. Es können spezielle Elemente für die Vorschau mit einem zusätzlichen Highlighting versehen werden. Diese Elemente müssen innerhalb des übergebene Texts mit XML-Tags umschlossen werden.

---

[35] (JetBrains Documentation Team 2016)

- **getAdditionalHighlightingTagToDescriptorMap**
  Sofern es für den Vorschau-Text spezielle Elemente mit einem zusätzlichen Highlighting gibt, wird hier eine Zuweisung von Element-Typ und Farbeinstellung für das Highlighting zurückgegeben.

[36] [37]

2.4.4.5 Erweiterungspunkt: languageInjector

Dieser Erweiterungspunkt kann verwendet werden, wenn der Punkt, an dem die Sprache eingefügt werden soll bereits ein „Language Injection Host“ ist, also das Interface ***PsiLanguageInjectionHost*** implementiert.

Für die Verwendung muss das Interface ***LanguageInjector*** implementiert werden, welches genau eine Methode ***getLanguagesToInject*** erwartet. Hier wird als Parameter ein ***PsiLanguageInjectionHost*** übergeben für den injizierte Dateien ermitteln werden sollen und eine Instanz von ***InjectedLanguagePlaces***, welches eine Registrierung darstellt, mit der neue injizierte Dateien für den übergebenen Host registriert werden können.

2.4.4.6 Erweiterungspunkt: multiHostInjector

Im Gegensatz zum Erweiterungspunkt ***languageInjector*** muss der Punkt, an dem die Sprache eingefügt werden soll, nicht bereits eine Implementierung von ***LanguageInjectionHost*** sein. Somit hat man die Möglichkeit, jedes beliebige PSI-Element zu einem Host zu machen solange man das entsprechende Interface implementiert. Zur Verwendung des Erweiterungspunkts, muss das Interface ***MultiHostInjector*** implementiert werden, welches folgende Methoden erwartet:

- **elementsToInjectIn**
  Muss eine Liste von ***PsiElement***-Klassen zurückliefern, für die der Injektor weitere Dateien injizieren kann.

- **getLanguagesToInject**
  Sofern ein PSI-Element eine der Klassen aus ***elementsToInjectIn*** implementiert, wird für das Element diese Methode aufgerufen. Hierbei gibt es den Parameter vom Typ ***MultiHostRegistrar***, an welchem die Injektion registriert werden kann. Als weiterer Parameter wird die Implementierung von ***PsiElement*** übergeben, für die die Registrierung durchgeführt werden kann.

---

[36] (JetBrains Documentation Team 2016)
[37] (JetBrains Development Team 2016)

Um eine Injektion bei der ***MultiHostRegistrar*** durchführen zu können, muss das ***PsiElement*** das Interface ***PsiLanguageInjectionHost*** implementieren. Dieses besteht aus folgenden Methoden:

- **isValidHost**
  Gibt zurück, ob diese Instanz Sprachinjektionen entgegen nehmen kann.

- **createLiteralTextEscaper**
  Muss eine Implementierung von ***LiteralTextEscaper*** zurückliefern, der auf Objekten arbeiten kann, die ***PsiLanguageInjectionHost*** implementieren. Dieser muss dafür sorgen, dass das Textformat des Hosts in das Format der injizierten Datei umgewandelt wird.

- **updateText**
  Wird aufgerufen, wenn sich der Text der injizierten Datei ändert. Es muss sichergestellt werden, dass der darunter liegende Host nach Aufruf der Methode die Änderungen übernommen hat. Dies stellt eine Umkehrfunktion zu der durch die Methode ***createLiteralTextEscaper*** gelieferten Implementierung dar.

Das Interface erbt von ***PsiElement***, also müssen auch alle Methoden von ***PsiElement*** implementiert werden. [38]

2.4.4.7 Erweiterungspunkt: toolWindow

Wenn ein neues Tool-Window in IntelliJ IDEA registriert werden soll, kann dieser Erweiterungspunkt verwendet werden. Neben einer eindeutigen ***id*** für das Tool-Window muss für die Angabe ***factoryClass*** das Interface ***ToolWindowFactory*** implementiert werden, welches die Methode ***createToolWindowContent*** erwartet. Hier wird neben dem aktuellen Projekt auch eine Instanz von ***ToolWindow*** bereitstellt, deren Inhalt befüllt werden kann.

Es können weitere optionale Angaben bei der Verwendung des Erweiterungspunkts gemacht werden:

- **anchor**
  Die Bildschirm-Seite an der der Button zum Öffnen des Tool-Windows dargestellt werden soll. Mögliche Werte: ***left***, ***right*** und ***bottom***

- **secondary**
  nicht weiter beschrieben

[38] (JetBrains Development Team 2016)

- **icon**
  Das Icon, welches für den Button zum Öffnen des Tool-Windows genommen werden soll. Muss eine Größe von 13x13 Pixeln haben.

- **conditionClass**
  nicht weiter beschrieben

- **canCloseContents**
  nicht weiter beschrieben

2.4.5 Action System

Durch das Action-System ist es einem Plugin möglich, weitere Menü- und Toolbar-Einträge in IDEA vorzunehmen. Hierzu muss die Klasse ***AnAction*** abgeleitet werden, da deren ***actionPerformed*** Methode aufgerufen wird, sobald eine Action ausgeführt werden soll.

Actions lassen sich in ein oder mehreren Gruppen zuteilen, welche wiederum Untergruppen haben können. So lassen sich Untermenüs abstrahieren. Jede Action und jede Action Group hat eine eindeutige ID, mit der sie identifiziert wird.

Mit Hilfe der Methode ***update*** werden Actions von IDEA aktualisiert. Dies passiert bei Untermenüs zum Beispiel bevor sie eingeblendet werden, damit die Actions noch vor ihrer Darstellung Einfluss nehmen können, ob und wie sie dargestellt werden sollen. Dies geschieht mit Hilfe eines Parameters vom Typ ***AnActionEvent***, welcher Informationen zum Kontext trägt, in dem die Aktion ausgeführt wurde. Dieser Parameter existiert auch für die zuvor genannte Methode ***actionPerformed***. [39]

2.5 Erweiterte Backus-Naur-Form

Die „Erweiterte Backus-Naur-Form“ ist eine Erweiterung der „Backus-Naur-Form“, welche eine formale Metasprache ist, um kontextfreie Grammatiken abzubilden.

Es gibt zwei Arten von Symbolen:

- Terminalsymbole
  Dies sind Zeichen, die in einem Text auftauchen können

- Nichtterminalsymbole
  Dies repräsentiert eine Reihe von Terminal- und Nichtterminalsymbole, die mit Hilfe von beliebig vielen Ausdrücken verbunden sind.

[39] (JetBrains Documentation Team 2016)

Die Zuordnung einer Reihe von Symbolen zu einem Nichtterminalsymbol nennt sich Produktionsregel. Das Ende einer Produktionsregel wird mit einem Semikolon bekannt geben.

Symbole können über folgende Ausdrücke in einer Produktionsregel verbunden werden:

- Komma bedeutet, dass sowohl das linke als auch das rechte Symbol zutreffen muss
- Der Schrägstrich bedeutet, dass entweder die linke Seite oder die rechte Seite zutreffen muss
- Die runde Klammer gruppiert einzelne Symbole
- Die Symbole in eckigen Klammern sind optional
- Von geschweiften Klammern umgebene Symbole können beliebig oft oder gar nicht auftreten

[40]

[40] (ISO/IEC 14977 1996)

3 Analyse und Design

Nachdem die Grundbegriffe, die für diese Ausarbeitung relevant sind, im vorherigen Kapitel erklärt wurden, findet darauf aufbauend in den nächsten Kapiteln eine Analyse des B.E.M. Patterns statt. Diese wird gefolgt von User-Stories, die die einzelnen Funktionen des zu erstellenden Plugins aus Benutzersicht beschreiben. Den Abschluss bildet eine Analyse der Anwendbarkeit der in den Grundlagen vorgestellten Plugin-API von IntelliJ IDEA für die beschriebenen User-Stories.

3.1 Analyse des B.E.M. Patterns

Das B.E.M. Pattern lässt sich wie in den Grundlagen beschrieben als einfaches Benennungsmuster anwenden. Dies allein sorgt bereits für eine bessere Struktur innerhalb eines Projekts. Es ist allerdings auch möglich, die Regeln ein wenig zu erweitern, um noch mehr von der verbesserten Struktur profitieren zu können. Das folgende Kapitel stellt einige Bezüge zur objektorientierten Programmierung in Java her. Darauf aufbauend und aufgrund von persönlichen Erfahrungen innerhalb von größeren Projekten definiere ich dann erweiterte Regeln, die man bei der Verwendung des B.E.M. Patterns in Betracht ziehen sollte.

3.1.1 Bezug zur objektorientieren Programmierung in Java

Mit Hilfe des B.E.M. Patterns lassen sich einige Aspekte der objektorientierten Programmierung aus der Java-Welt in CSS Regeln etablieren:

- Ein Block entspricht einer Klasse, welche einen DOM-Knoten als Referenz erwartet, um auf ihm CSS-Styles anzuwenden.

- Die Menge der Elemente eines Blocks wiederum entsprechen Attributen einer Klasse, welche mit Referenzen zu DOM-Knoten belegt werden. Die Klasse steuert ebenfalls die CSS-Styles der referenzierten DOM-Knoten.

- Die Menge der Modifier eines Blocks sind Ableitungen der ursprünglichen Block Klasse. Es werden also alle CSS-Styles für den Block und dessen Elemente des ursprünglichen Blocks vererbt und die Ableitung kann CSS-Styles ändern oder neue hinzufügen. Durch Einschränkungen der CSS-Sprache lässt sich ein vererbter Style allerdings nicht entfernen, sondern lediglich überschreiben (vgl. Kapitel 2.1.3.3).
  Dies entspricht also dem Prinzip der Vererbung (vgl. 2.2.1.7).

- Durch das Anbringen eines Blocks an einen DOM-Knoten in Form eines Klassennamens werden die Style-Regeln angewandt.
  Dies entspricht der Instanziierung der Block Klasse, welche im Konstruktor eine Referenz auf den DOM-Knoten erhält und die entsprechende Operation durchführt.

- Das Anbringen eines Block Klassennamens in Kombination mit einem Modifier Klassennamen entspricht analog dazu der Instanziierung der spezielleren Modifier-Klasse, die die Block-Klasse erweitert.

- Auch Datenkapselung (vgl. Kapitel 2.2.1.3) spielt eine wichtige Rolle. Durch das Vergeben von Klassennamen in CSS wird häufig auf Style-Regeln direkt an einem DOM-Element verzichtet. Weitet man dies sogar so aus, dass man sagt, es darf auf DOM-Ebene nur mit CSS Klassen gearbeitet werden, kapselt dies die Implementierung für alle Styles der einzelne Komponenten, sodass man nur noch über Schnittstellen in Form von Klassennamen darauf zugreifen kann.

- Auch Polymorphie (vgl. Kapitel 2.2.1.6) spielt eine Rolle. Setzt man dieselbe Klasse an durch CSS Selektoren unterscheidbare DOM-Elemente (z.B. weil sie unterschiedliche Tag-Namen haben), kann sie durch unterschiedliche Selektoren ein jeweils anderes Verhalten bei der jeweiligen Verwendung hervorrufen.

Diese Aspekte erhöhen die Modularität von CSS Regeln und erlauben es, Webanwendungen besser zu strukturieren. Das B.E.M. Pattern kann also nicht nur wie in Kapitel 2.3.1 beschrieben als Namenspattern, sondern auch als Strukturpattern angesehen werden.

3.1.2 Erweiterung der B.E.M. Regeln

Das B.E.M. Pattern ist in seiner Definition sehr offen gehalten. Zur Verwendung in großen Projekten haben sich aus persönlicher Erfahrung striktere Regelungen bewährt, die ich in dieser Ausarbeitung als „Stricter B.E.M.“ bezeichne.

3.1.2.1 Verschiedene B.E.M. Entitäten in CSS Selektoren

Das B.E.M. Pattern dient der besseren Strukturierung von CSS-Klassennamen und –Regeln (vgl. Kapitel 2.3.1). Ein Block repräsentiert hierbei eine Komponente. Da ein Qualitätsmerkmal für gute Architektur die lose Kopplung zwischen den Komponenten ist, soll dies auch für Frontend Komponenten gelten.

```
.message__text {
  background-color: #ffffff;
  color: #000;
}

.main-content .message__text {
  background-color: #efefef;
}

.main-content .top .message__text {
  background-color: #dcdcdc;
}

.header .message__text,
.footer .message__text{
  background-color: #000;
  color: #fff;
}
```

Abbildung 5: Starke Kopplung von B.E.M. Blöcken

Eine starke Kopplung aus B.E.M. Sicht würde es zum Beispiel bedeuten, wenn eine CSS Regel wie in Abbildung 5 mehrere B.E.M. Blöcke in Abhängigkeit bringt. Hierdurch lassen sich bestimmte Ausprägungen der Liste nicht mehr unabhängig von anderen Komponenten benutzen, der DOM muss also für alle anderen Ausprägungen sicherstellen, dass sich entsprechende Eltern-Knoten mit der richtigen Klasse vor den DOM-Knoten, die die Liste repräsentieren, befinden. Außerdem findet sich eine Dopplung von Regeln wieder, die dasselbe Aussehen in verschiedenen Kontexten wiederverwenden möchten.

Interessant wird es insbesondere dann, wenn ein Modifier ***highlighted*** hinzugefügt werden soll, der zum Beispiel dafür sorgt, dass sich eine Liste besonders hervorhebt. Hierfür würde der CSS-Code aus der folgenden Abbildung hinzugefügt werden müssen:

```
.message--error .message__text,
.main-content .message--error .message__text,
.main-content .top .message--error .message__text {
  color: #ff0000;
}

.header .message--error .message__text,
.footer .message--error .message__text {
  background-color: #ff0000;
  color: #ffffff;
}
```

Abbildung 6: Konsequenzen von starker Kopplung in B.E.M.

Es ist eine Dopplung der CSS Regeln mit tiefen Selektoren erfolgt, die anschließend optimiert wurden, indem Regeln mit gleichen Deklarationen zusammengefasst wurden. Während man den zweiten Selektor in der ersten Regel theoretisch weglassen kann, da durch die Kaskade (vgl. Kapitel 2.1.3.4) bereits sichergestellt ist, dass die Deklarationen des Modifiers ***error*** auch die Deklarationen für den ***message*** Block unter ***content-main*** überschreiben, ist es durch die Spezifität des Selektors, der im Kontext ***main-content*** und ***top*** die Schriftfarbe ändert, notwendig, einen weiteren Selektor zu schreiben.
Obwohl der ***message*** Block unter dem Block ***header*** und ***footer*** gleich aussieht, müssen auch hier noch zwei Selektoren erstellt werden, damit der Modifier ***error*** entsprechend das Aussehen von ***message*** in beiden Kontexten beeinflusst.

Wie man auch schon an der Beschreibung der Regeln im letzten Absatz erkennen kann, wird es immer schwieriger, Zusammenhänge auszudrücken und Zuständigkeiten der Blöcke zu verstehen.

Eine Umbenennung einer der Kontexte (z.B. ***main-content***) würde außerdem dazu führen, dass Regeln, die sich auf ***message*** Blöcke beziehen, mit angepasst werden müssen.

```
.message__text {
  background-color: #ffffff;
  color: #000;
}

.message--main .message__text {
  background-color: #efefef;
}

.message--top .message__text {
  background-color: #dcdcdc;
}

.message--inverted .message__text {
  background-color: #000;
  color: #fff;
}

.message--highlighted.message--inverted .message__text {
  background-color: #ff0000;
}
```

Abbildung 7: Lose Kopplung von B.E.M. Blöcken

Der in Abbildung 7 befindliche Code zeigt, wie man die kontextabhängigen Styling-Deklarationen von Blöcken durch Modifier ersetzen kann. Während es unter dem Aspekt der Regel Spezifität für den nun eingesetzten Modifier ***main-content*** keine nennenswerten Vorteile gibt, kann durch den Modifier ***top*** die Spezifität so verringert werden, dass eine Dopplung des Selektors für ***top*** bei Verwendung des Modifiers ***error*** nicht mehr notwendig ist.

Der Modifier ***inverted*** zeigt außerdem, wie man durch Extraktion von Gemeinsamkeiten unter einem gemeinsamen Namen einen allgemeingültigen, wiederverwendbaren Modifier erzeugt, bei dem durch den Namen bereits deutlich wird, was er macht, nämlich die Farben des nicht modifizierten Blocks zu invertieren.

Die Kombination von ***inverted*** und ***error*** muss nun lediglich die Hintergrundfarbe des Textes anpassen, weil die Reihenfolge der CSS-Regeln effektiv genutzt werden kann, um die Vorrangigkeit von Farben zu beschreiben und der weiße Text bereits definiert ist.

Eine lose Kopplung kann durch folgende Regel unterstützt werden:

B.E.M. Klassennamen für verschiedene Blöcke dürfen nicht in der gleichen CSS-Regel verwendet werden.

3.1.2.2 Position der B.E.M. Entitäten in CSS-Selektoren

Kapitel 2.3.1 beschreibt die Einschränkungen, die sich bei der Verwendung von B.E.M. Entitäten innerhalb des HTML Dokuments ergeben. Dies lässt sich auch für CSS-Regeln anwenden:

1) ***Wenn in einer Regel ein Klassen-Selektor für einen B.E.M. Block oder B.E.M. Modifier verwendet wird, so dürfen Klassen-Selektoren für B.E.M. Elemente desselben Blocks nur verwendet werden, wenn diese sich auf Nachfahren beziehen.***

2) ***Wenn in einer Regel ein Klassen-Selektor für einen B.E.M. Block verwendet wird, so dürfen Klassen-Selektoren für B.E.M. Modifier desselben Blocks nur verwendet werden, wenn diese sich auf den gleichen DOM-Knoten beziehen.***

```
.box__title .box,
.box.box__title {
  /* ... */
}

.box .box--special,
.box--special .box,
.box ~ .box--special {
  /* ... */
}
```

Abbildung 8: Ungültige B.E.M. Regeln

Die obige Abbildung zeigt Verstöße gegen die zuvor genannten Regeln für CSS Selektoren.

Im ersten Regelsatz befinden sich Regeln, die gegen 1) verstoßen:

- Das B.E.M. Element ***title*** wird vor dem B.E.M. Block ***box*** erwartet
- Das B.E.M. Element ***title*** soll sich am gleichen DOM-Knoten wie der B.E.M. Block ***box*** befinden

Analog dazu zeigt der zweite Regelsatz Verstöße gegen 2):

- Der Modifier ***special*** soll sich an einem Nachfahren des DOM-Knoten befinden, der den Block ***box*** trägt.
- Wie zuvor, nur, dass hier ein Vorfahre in Betracht gezogen wird.
- Es wird ein Geschwister-Knoten ***special*** für den DOM-Knoten gesucht, der den Block ***box*** trägt.

Die Einführung der zusätzlichen Regel trägt dazu bei, das B.E.M. Pattern richtig einzusetzen:

Durch die Definition der Regel wird zum einen dazu beigetragen, dass sich die bereits für die DOM-Struktur definierte Regel auch im CSS wiederspiegelt, also die einzelnen B.E.M. Entitäten auch nur mit der richtigen Relation zu anderen Entitäten entsprechende Style-Definitionen bekommen.

Außerdem wird etwas verhindert, was ich schon in vielen Projekten gesehen habe:
Ein Block wird mit Hilfe einer komplexen CSS-Regel in Abhängigkeit zur Verschachtelungstiefe zu sich selbst mit anderen Styling-Definitionen belegt.

```
.list__item {
  margin-left: 10px;
}

.list .list .list__item {
  margin-left: 8px;
}

.list .list .list .list__item {
  margin-left: 6px;
}
```

Abbildung 9: B.E.M. Anti-Pattern Nesting

Die obere Abbildung zeigt eine aus meiner Sicht falsche Verwendung des B.E.M. Patterns. Es soll erreicht werden, dass die Elemente einer Liste in einer tieferen Verschachtelungsebene weniger stark eingerückt werden sollen. Dies wird mit Hilfe von tieferen Selektoren erzielt.

Würde man jetzt einen Modifier ***no-indention*** einführen wollen, so müsste er folgendermaßen definiert werden:

```
.box--no-indent {
  margin-left: 0;
}

.box .box--no-indent {
  margin-left: 0;
}

.box .box .box--no-indent {
  margin-left: 0;
}
```

Abbildung 10: Konsequenzen des Anti-Patterns

Es müssen drei zusätzliche Regeln definiert werden, weil sonst die Stärke Regelstärke des Modifiers nicht ausreicht, um die Definitionen für die tieferen Verschachtelungen zu überschreiben (vgl. Kapitel 2.1.3.4). Was in diesem einfachen Beispiel noch relativ unerheblich ist, verstärkt sich durch die Menge an Style-Deklarationen und die beeinflussten Elemente und macht die CSS Regeln des Blocks sehr unübersichtlich.

Da das B.E.M. Pattern genau diese Tiefe Verschachtelung vermeiden soll, ist es besser, das, was durch die Verschachtelung abgebildet werden soll, stattdessen ebenfalls durch einen Modifier abzubilden, wodurch die Stärke des B.E.M. Patterns besser genutzt wird:

```
.box__item {
  margin-left: 10px;
}

.box--level-2 .box__item {
  margin-left: 8px;
}

.box--level-3 .box__item {
  margin-left: 6px;
}

.box--no-indent .box__item {
  margin-left: 0;
}
```

Abbildung 11: Bessere Anwendung des B.E.M. Patterns

3.1.2.3 **Modifizierte Elemente**

Für die Relation von Blöcken und Elementen hinsichtlich ihrer Position im DOM gilt: *„Element Klassennamen dürfen nur in Nachfahren des DOM-Knotens verwendet werden, welche den jeweiligen Block Klassennamen trägt.“* (vgl. Kapitel 2.3.1).

Dies hat Auswirkungen auf den CSS-Selektor, der verwendet werden muss, um Deklarationen für ein Element in Abhängigkeit zu einem Modifier desgleichen Blocks zu definieren. Das Element kann ein beliebiger Nachfahre sein und somit wird sein Klassen-Selektor mit Hilfe des Nachfahren-Selektors (vgl. Kapitel 2.1.3.2) mit dem Klassen-Selektor des Modifiers kombiniert. Die nachfolgende Abbildung zeigt einen typischen Anwendungsfall für modifizierte Elemente:

```
.list {
  display: block;
  padding: 5px;
  margin: 0;
  list-style: none;
}

.list__item {
  margin-left: 20px;
}

.list--top-level {
  border: 1px solid black;
}

.list--top-level .list__item {
  margin-left: 0;
}
```

Abbildung 12: Code-Beispiel für ein modifiziertes Element mit Hilfe des Nachfahren-Selektors

Hier sollen die ***item*** Elemente eines ***list*** Blocks nach rechts eingerückt werden. Bei Verwendung des Modifiers ***top-level*** soll die Einrückung entfallen.

Nutzt man den Block ***list*** ohne und mit Modifier in jeweils eigenen Unterbäumen des DOM, verhält sich alles wie erwartet. Ein Problem entsteht jedoch, wenn Listen verschachtelt dargestellt werden sollen.

```
<ul class="list list--top-level">
  <li class="list__item">
    Chapter 1
    <ul class="list">
      <li class="list__item">Chapter 1.1</li>
      <li class="list__item">Chapter 1.2</li>
    </ul>
  </li>
  <li class="list__item">
    Chapter 2
    <ul class="list">
      <li class="list__item">Chapter 2.1</li>
      <li class="list__item">Chapter 2.2</li>
    </ul>
  </li>
</ul>
```

Abbildung 13: Nutzung des Blocks "list" in verschachtelten Listen

Die obere Abbildung zeigt die Nutzung des Blocks ***list*** in einer verschachtelten Liste, welche ein Inhaltsverzeichnis abbilden soll. Der Modifier ***top-level*** wird hierbei verwendet, um die erste Ebene der Liste nicht einzurücken, während alle anderen Kapitel-Ebenen eingerückt werden sollen.

Während die Nutzung des Blocks aus B.E.M. Sicht gültig ist, entspricht die Darstellung nicht der Erwartung: Nicht nur die ***item*** Elemente des ***list*** Blocks mit dem Modifier ***top-level*** sind nicht eingerückt sondern auch alle anderen ***item*** Elemente.

Man kann diesen Effekt logisch erklären (vgl. hierfür Kapitel 3.1.1):

- Das Element ***item*** wird als DOM Referenz an alle Block Instanzen übergeben, die durch Vorfahr-Knoten erzeugt wurden.
- Eine der Block-Instanzen nutzt die Basis-Klasse des Blocks ***list***, die andere hingegen die durch den Modifier ***top-level*** entstandene Ableitung.
- Beide Instanzen übertragen ihre Deklarationen an den referenzierten DOM-Knoten.

Es besteht also eine mehrfache Zugehörigkeit von B.E.M. Elementen hinsichtlich ihrer Block-Instanz. Während dies für die Basis-Klasse zu keinen sichtbaren Seiteneffekten führt, weil diese von jedem Modifier des Blocks abgeleitet wird, sorgt die Verwendung des Modifiers dafür, dass alle B.E.M. Elemente, die in Nachfahren des ursprünglichen DOM-Knotens, an dem der Modifier angebracht wurde, ebenfalls die Deklarationen erhalten.

Um dies zu verhindern, muss die Zugehörigkeit weiter spezifiziert werden. Dies gelingt mit Hilfe des Kind-Selektors (vgl. Kapitel 2.1.3.2). Die folgende Abbildung zeigt die entstandene Abwandlung des CSS-Codes, mit dessen Hilfe das in Abbildung 13 gezeigte Anwendungs-Beispiel so funktioniert, wie es erwartet werden würde.

```
.list {
  display: block;
  padding: 5px;
  margin: 0;
  list-style: none;
}

.list__item {
  margin-left: 20px;
}

.list--top-level {
  border: 1px solid black;
}

.list--top-level > .list__item {
  margin-left: 0;
}
```

Abbildung 14: Code-Beispiel für ein modifiziertes Element mit Hilfe des Nachfahren-Selektors

Um unerwünschte Seiteneffekte bei Verwendung des gleichen B.E.M. Blocks mit unterschiedlichen Modifiern auf dem gleichen DOM Pfad zu verhindern, lässt sich also eine der in Kapitel 3.1.2.2 definierten Regeln noch weiter spezifizieren:

„Wenn in einer Regel ein Klassen-Selektor für einen B.E.M. Block oder B.E.M. Modifier verwendet wird, so dürfen Klassen-Selektoren für B.E.M. Elemente desselben Blocks nur verwendet werden, wenn diese sich auf Kind-Knoten beziehen."

3.2 Randbedingungen

Das Plugin soll beim Einsatz des B.E.M. Patterns unterstützen ohne dabei andere Funktionalitäten der IDE einzuschränken oder auszuschließen.

Wie in Kapitel 2.6.2 beschrieben, gibt es die Möglichkeit bei der Verwendung der IntelliJ IDEA Community Edition eine eigene Sprache für die Entwicklungsumgebung zu entwickeln oder bei Verwendung der IntelliJ IDEA Ultimate Edition eine Erweiterung für die existierende CSS Sprache zu schreiben.

Würde man obige Anforderung mit der Community Edition erfüllen wollen, so wäre es notwendig, einen eigenen CSS Parser zu entwickeln, alle bestehenden Funktionalitäten der IDE nachzubauen und danach mit der Entwicklung der eigenen Funktionalitäten zu beginnen. Kompatibilität mit anderen Plugins ist hier nicht gewährleistet.

Aufgrund dieser Einschränkungen wird sich die Implementierung auf die alternative Variante - also der Erweiterung der in der Ultimate Edition liegenden CSS Sprache - stützen.

3.3 User Stories

Im Folgenden werden alle für das Plugin geplanten Funktionalitäten in Form von User-Stories beschrieben.

Es gibt zwei wichtige Personas: Sally und Adam.

Sally (26) ist eine Frontend-Entwicklerin, die in einem großen Team an diversen Projekten mit komplexem CSS Code arbeitet. Die in ihrem Team verwendete Entwicklungsumgebung ist IntelliJ IDEA. Sie hat sehr gute Kenntnisse in HTML und CSS.

Bedingt durch die steigende Größe der Projekte wurde vor einigen Wochen von den Frontend-Architekten des Teams beschlossen, das B.E.M. Pattern als Struktur- und Namenspattern neu zu etablieren. Da Sally allerdings noch nicht viel Erfahrung mit dem Pattern gesammelt hat, ist sie unsicher, ob sie das neue Pattern richtig verwendet. Zudem ist sie noch nicht von den Vorteilen dieser sehr befremdlich wirkenden CSS Benennung überzeugt.

Adam (30) ist einer der Frontend-Architekten in Sallys Team. Er beschäftigt sich viel mit neuen Technologien und hat es sich zum Hauptziel gemacht, dass möglichst modulare Projekte, mit viel wiederverwendbarem Quellcode entstehen. Zu seinen Fachgebieten gehören neben HTML und CSS auch die objektorientiere Programmierung in Java.

Vor kurzer Zeit hat er einen Artikel über das B.E.M. Pattern im „Smashing Magazine“ gelesen und geprüft, ob man das Pattern bei ihm im Team anwenden kann und welche Vorteile es bringen könnte. Da es sein Hauptziel gut unterstützt, hat er mit seinen anderen Frontend-Kollegen beschlossen, das neue Pattern für alle laufenden Projekte zu etablieren.
Er erhofft sich, einen besseren Überblick über bestehende Frontend Module zu gewinnen und diese klarer voneinander abgrenzen zu können. Wie alle anderen Team-Mitglieder, verwendet auch Adam IntelliJ IDEA.

3.3.1 User Story 1: Syntax-Highlighting für B.E.M.

Um Tipp-Fehler bei der Verwendung des B.E.M. Patterns zu vermeiden, möchte Sally während der Bearbeitung von CSS Regeln und Verwendung von CSS Klassen im Text-Editor von IntelliJ IDEA sehen, bei welchen CSS Klassennamen es sich um Blöcke, Elemente oder Modifier handelt.

3.3.1.1 **Anforderungen**

- Alle CSS Klassennamen, die dem B.E.M. Pattern entsprechen sollen sich in ihrer Darstellung im Editor von Klassennamen unterscheiden, die dem B.E.M. Pattern nicht entsprechen.
- Es soll leicht zwischen Blöcken, Elementen und Modifiern unterschieden werden können, hierzu sollen sich einzelnen Teile einer B.E.M. Entität unterschiedlich darstellen:
    - Block Bezeichner
        - Klare Kennzeichnung als Hauptbezeichner
    - Element Separator
        - Als syntaktisches Element in den Hintergrund gerückt
    - Modifier Separator
        - Ebenfalls in den Hintergrund gerückt
    - Element Bezeichner
        - Hervorgehoben als Nebenbezeichner, aber von Modifier Bezeichnern klar zu unterscheiden
    - Modifier Bezeichner
        - Hervorgehoben als Nebenbezeichner, aber von Element Bezeichnern klar zu unterscheiden

3.3.1.2 **Abgrenzung**

Das Syntax-Highlighting soll sich auf CSS Klassen, die in CSS Regeln verwendet werden beschränken.

3.3.1.3 **Beispiel**

```
.other__naming__pattern {
    display: none;
}

.grid {
    display: block;
    background-color: #919191;
}

.grid--2-columns {
    background-color: #BF8080;
}

.grid__row {
    display: block;
    font-size: 0;
}

.grid__col1,
.grid__col2 {
    display: inline-block;
    font-size: 1rem;
}

.grid__col1 {
    width: 100%;
}

.grid--2-columns .grid__col1 {
    width: 50%;
}

.grid--2-columns .grid__col2 {
    width: 100%;
}
```

Abbildung 15: Syntax-Highlighting in CSS Regeln

Wie in Abbildung 15 zu sehen ist, wird der Block Bezeichner ***grid*** immer unterstrichen dargestellt und schwarz. Bei der CSS Klasse ***other__naming__pattern*** handelt es sich um einen nicht B.E.M. konformen Namen, der also auch nicht als Block, Element oder Modifier erkannt werden soll. Der Modifier Bezeichner ***special*** und die Element Bezeichner ***row*** und ***cell*** werden in jeweils anderen Farben dargestellt. Die Separatoren für Elemente und Modifier werden entsprechend ausgegraut.

3.3.2 User Story 2: Farbeinstellungen für das B.E.M. Syntax-Highlighting

Während Adam mit den Standard-Einstellungen für das Syntax-Highlighting sehr gut arbeiten kann, möchte Sally eine Möglichkeit haben, die Farbeinstellungen für das Syntax-Highlighting nach ihren Bedürfnissen anzupassen. Existierende Funktionalitäten kennt sie bereits für das Ändern der Darstellung von CSS Elementen aus dem CSS Plugin, daher wünscht sie sich, dass ein analoger Mechanismus zur Verfügung gestellt wird.

3.3.2.1 **Anforderungen**

- Es gibt unter ***Settings*** – ***Editor*** – ***Colors & Fonts*** einen neuen Eintrag ***B.E.M.***
- Der Aufbau des darin befindlichen Panels ist analog zu dem des Eintrags ***CSS***
- Für folgende Einträge kann die Darstellung individuell angepasst werden:
    - Block Bezeichner
    - Element Bezeichner
    - Modifier Bezeichner
    - Element Separator
    - Modifier Separator
    - Nicht B.E.M. konforme Klassennamen

3.3.3 User Story 3: B.E.M. Tool-Window

Adam hat den Eindruck, dass im Laufe der Zeit sehr viele komplexe B.E.M. Blöcke entstanden sind, die man in mehrere kleine Blöcke aufteilen könnte. Die reine Betrachtung der CSS Regeln stellt für ihn keine wirklich effiziente Art dar, um einen Überblick über die B.E.M. Struktur des Projekts zu gewinnen.

Daher wünscht er sich, dass es eine Möglichkeit in der IDEA gibt, sich eine nach Blöcken aufgeteilte B.E.M. Aggregation der aktuellen Datei anzuzeigen, die es möglich macht, strukturelle Probleme zu analysieren. Würde er beispielsweise sehen können, dass einige Elemente nur unterhalb von Modifiern verwendet werden, so könnte er Kandidaten für einen neuen, unabhängigen Block identifizieren. Dies würde seiner Ansicht nach die Wiederverwendbarkeit einzelner B.E.M. Module innerhalb des Projektes deutlich erleichtern können.

3.3.3.1 **Anforderungen**

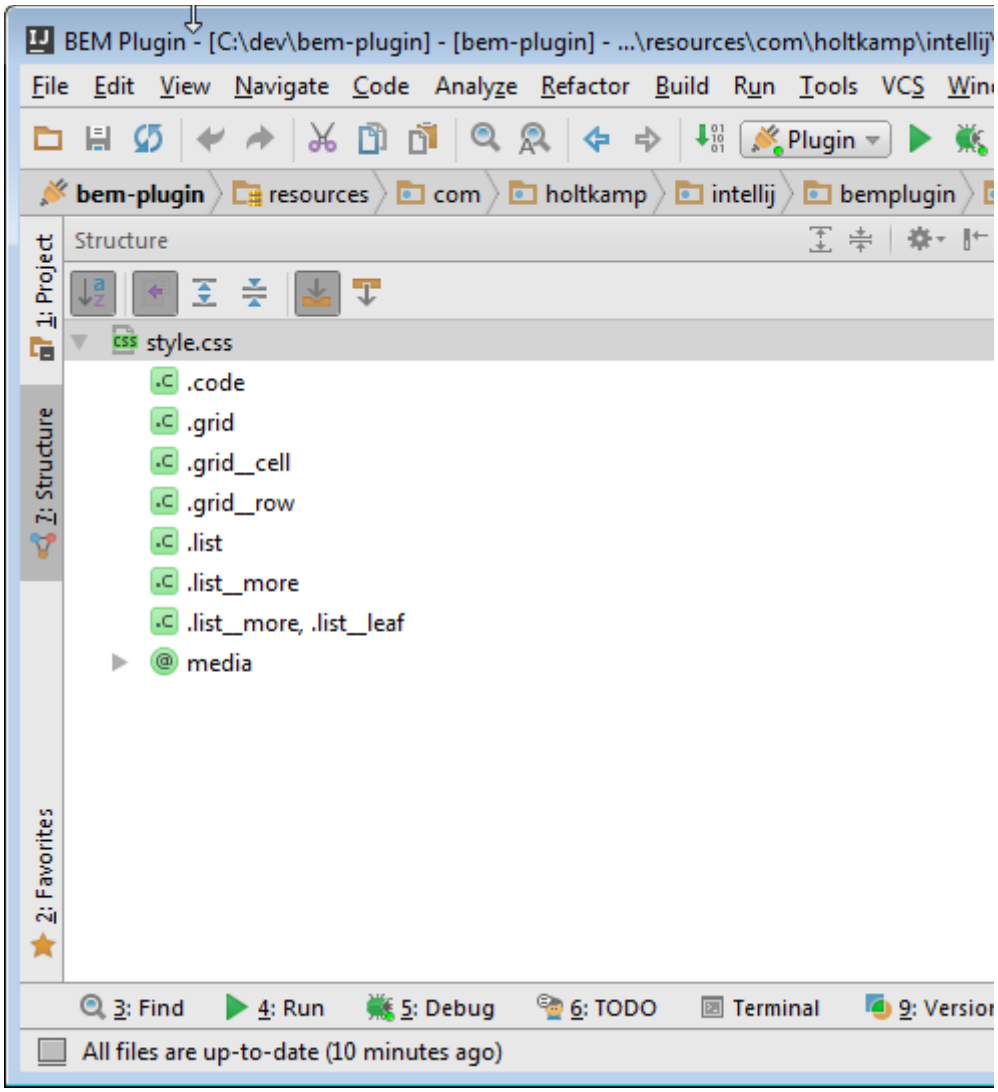


Abbildung 16: Tool-Windows

- Es gibt einen Toggle-Button ***B.E.M.*** in der linken unteren Toolbar über dem Button ***Favorites*** (vgl. Abbildung 16) der das B.E.M. Tool-Window ein- und ausschalten kann.
- Der Haupt-Inhalt des Tool-Windows ist ähnlich aufgebaut wie die „Structure View“ und wird ebenfalls an die aktuell geöffnete Datei gebunden.
- Der Rootknoten des Trees soll die aktuelle Datei darstellen
- Darunter soll eine übersichtliche Auflistung der aggregierten B.E.M. Entitäten erfolgen, hierzu gehören:
    - Blöcke

    - Elemente
    - Modifier
    - Modifizierte Elemente (Also Elemente, die durch einen Modifier verändert werden)
    - Implizite Entitäten (mehr dazu in den Beispielen)
- Es soll klar zu erkennen sein, um welche Art von Entität es sich handelt.
- Ein Indikator soll anzeigen, ob es CSS Regeln zu den aufgeführten Entitäten gibt.
- Es gibt eine Toolbar mit folgenden Funktionen:
    - Sortierung der B.E.M. Entitäten nach Name (wenn nicht aktiviert, bestimmt die Reihenfolge in Datei die Ordnung)
    - Alle Knoten ausklappen
    - Alle Knoten zuklappen

3.3.3.2 **Abgrenzung**

- Sofern sich eine dynamische Aktualisierung des B.E.M. Baums bei Änderung der geöffneten CSS Datei als zu komplex herausstellt oder sich negativ auf die Performance von IDEA auswirkt, kann die Ansicht auch mit Hilfe einer zusätzlichen ***Refresh*** Funktion in der Toolbar des B.E.M. Tool-Windows erfolgen.

3.3.3.3 **Beispiele**

```
.box {
    width: 100%;
    border: 1px solid black;
}

.box--highlighted {
    border-color: red;
}

.box__header {
    background-color: gray;
}

.box__content {
    background-color: white;
    color: #000;
}

.box__footer {
    background-color: gray;
}

.box--highlighted .box__content {
    background-color: red;
    color: #fff;
}

.box--content-only .box__header,
.box--content-only .box__footer {
    display: none;
}

.example--bad .example__text {
    font-style: italic;
}
```

Abbildung 17: Beispiel zu aggregiertem B.E.M.

Aus Abbildung 17 kann man B.E.M. Entitäten aggregieren und bereits markieren, welche der Entitäten CSS Regeln haben, die auf sie zutreffen, und welche nur als Hilfskonstrukt zur Selektion einer anderen Entität eingeführt wurden, also implizit dazugekommen sind (grau markiert):

- Block: box
    - Element: header
    - Element: content
    - Element: footer
    - Modifier: highlighted
        - Modifiziertes Element: content
    - Modifier: content-only
        - Modifiziertes Element: header
        - Modifiziertes Element: footer
- Block: example
    - Modifier: bad

        - Modifiziertes Element: text

Hieraus wiederum kann man weitere implizite Elemente und implizite modifizierte Elemente erkennen, welche in keiner CSS Regel explizit erwähnt werden, aber logische B.E.M. Entitäten aus den ermittelten Informationen darstellen (grau markiert und violett hinterlegt):

- Block: box
    - Element: header
    - Element: content
    - Element: footer
    - Modifier: highlighted
        - Modifiziertes Element: header
        - Modifiziertes Element: content
        - Modifiziertes Element: footer
    - Modifier: content-only
        - Modifiziertes Element: header
        - Modifiziertes Element: content
        - Modifiziertes Element: footer
- Block: example
    - Element: text
    - Modifier: bad
        - Modifiziertes Element: text

3.3.4 User Story 4: Generierung von B.E.M. Entitäten

Bei ihrer täglichen Arbeit bemerkt Sally, dass sie häufig mit Hilfe von „Kopieren und Einfügen“ Block Bezeichner aus ihrem CSS Code in Selektoren überträgt, um neue B.E.M. Entitäten zu erstellen. Auch beim Anlegen von neuen, modifizierten Elementen kopiert sie meistens zunächst eine vorhandene Regel des Elements, um dort eine Modifier CSS-Klasse voranzustellen.

Um effektiver arbeiten zu können, wünscht sie sich daher, dass es in IDEA die Möglichkeit gibt, ausgehend von einer vorhandenen B.E.M. Entität, weitere Entitäten generieren zu können. Hierbei sollen möglichst viele Informationen bereits mit sinnvollen Vorbelegungen versehen sind.

Auch Adam sieht den Vorteil dieser Funktionalität und erinnert sich, dass man in Java-Klassen Attribute und Methoden automatisch generieren lassen kann und hierbei bereits seitens der IDE sinnvolle Vorschläge gemacht werden, die durch Analyse des existierenden Codes entstehen.

3.3.4.1 **Anforderungen**

- Bei der Bearbeitung einer CSS Datei kann durch Aufruf des Menüeintrags ***Code*** - ***Generate*** (***Alt*** + ***Einfg***) ein Menueintrag: ***Create B.E.M. Block*** ausgewählt werden.
- Dies erzeugt ausgehend von der aktuellen Cursor-Position im Text-Editor an der nächsten gültigen Stelle eine CSS Regel für einen B.E.M. Block, wobei der Block Bezeichner mit einem Platzhalter versehen und vorselektiert ist, sodass die nächste Eingabe ihn entsprechend überschreiben kann.
- Die Verwendung der ***Return*** Taste schließt die Generierung des Blocks ab und setzt den Cursor zwischen die geschweiften Klammern, sodass direkt mit der Eingabe von CSS Attributen begonnen werden kann.
- Die Verwendung der ***Esc*** Taste entfernt alle Platzhalter und schließt die Generierung mit der aktuellen Belegung aller Platzhalter ab.
- Es soll analoge Funktionalitäten für B.E.M. Elemente, Modifier und modifizierte Elemente geben:
    - Element: Nach der Eingabe des Block Bezeichners muss noch der Element Bezeichner eingegeben werden
    - Modifier: Nach der Eingabe des Block Bezeichners muss noch der Modifier Bezeichner eingeben werden
    - Modifiziertes Element: Nach der Eingabe des Block Bezeichners wird der Modifier Bezeichner eingeben und danach der Element-Bezeichner.
- Bei Ausführung der Funktionalität aus einer bestehenden CSS Regel oder eines Regelsatzes heraus, sollen die Platzhalter der verschiedenen Bezeichner so weit wie möglich mit den Informationen, die sich aus einer Aggregation der B.E.M. Entitäten ergeben, vorbelegt sein, sodass man diese ohne sie zu überschreiben, durch betätigen der ***Return*** Taste übernehmen kann.
- Sind die Informationen mehrdeutig, gewinnt die zuerst definierte, da dies immer noch einen größeren Mehrwert darstellt, als einen Platzhalter zu verwenden, der auf jeden Fall überschrieben wird.
- Alle Code-Generierungen sollen mit Hilfe des Menüeintrags ***Edit*** – ***Undo*** wieder rückgängig gemacht werden können.

3.4 Anwendung der Plugin-API

Wie im Kapitel Plugin Entwicklung beschrieben, erfolgt die Entwicklung von Plugins in IDEA durch die Nutzung von existierenden Erweiterungspunkten (vgl. Kapitel 2.4.4) und der Definition von Actions (vgl. Kapitel 2.4.5).

Die folgenden Unterkapitel beschreiben, welche Mittel für die einzelnen Funktionalitäten des Plugins verwendet werden sollen. Hierbei bildet die Zahl in der Überschrift der Anwendung die jeweilige User-Story ab, für die die Plugin API angewendet werden soll.

3.4.1 Plugin Kern

Der Kern des B.E.M. Plugins besteht aus der Definition der Sprache und deren Einbindung, auf denen alle weiteren Features aufbauen.

3.4.1.1 Sprachdefinition

Eine neue Sprache kann IntelliJ über die Einbindung einer Parser Definition bekannt gemacht werden. Hierzu wird der Erweiterungspunkt ***lang.parserDefinition*** aus dem Language Support verwendet (vgl. Kapitel 2.4.4.2).

Um einen Parser und einen Lexer schreiben zu können, benötige ich zunächst eine Grammatik für das B.E.M. Pattern. Hierzu stütze ich mich auf die Definition des Tokens IDENT, welches die CSS2 .1 Spezifikation in Flex Notation hinterlegt hat.

```
ident=[-]?{nmstart}{nmchar}*
nmstart=[_a-z]|{nonascii}|{escape}
nonascii=[^\0-\237]
escape={unicode}|\\[^\n\r\f0-9a-fA-F]
unicode=\\[0-9a-f]{1,6}(\r\n|[ \n\r\t\f])?
nmchar=[_a-z0-9-]|{nonascii}|{escape}
nl=\n|\r\n|\r|\f
w=[ \t\r\n\f]*
```

Abbildung 18: CSS IDENT in Flex Notation

Die obere Abbildung zeigt die für das Ident Token relevanten Makros. Es sei angemerkt, dass das RFC darauf hinweist, dass der Tokenizer case-insensitive definiert ist, somit fällt die Unterscheidung zwischen Klein- und Großbuchstaben weg.

Für die Erzeugung einer Grammatik für das B.E.M. Pattern lege ich - basierend auf der B.E.M. Definition (vgl. Kapitel 2.3.1) - folgende Regeln fest:

- Wenn ein IDENT Token eine B.E.M. Entität darstellt, so erhält sie mindestens einen Block-Bezeichner.
- Der Block-Bezeichner ist immer der erste Teil der B.E.M. Entität und niemals leer.
- Ein IDENT Token, das ein B.E.M. Element repräsentiert, muss einen Block-Bezeichner beinhalten, der ebenfalls ein gültiges IDENT Token repräsentieren kann.
- Gleiches gilt für IDENT Token, die ein B.E.M. Modifier repräsentieren.
- Ein Element Bezeichner ist niemals leer

- Ein Modifier Bezeichner ist niemals leer

- Einzelne B.E.M. Entitäten werden durch Whitespaces getrennt

Die folgende Abbildung zeigt die Grammatik, die B.E.M. Entitäten akzeptiert, in der erweiterten Backus-Naur-Form.

```
Entities = EntityStart, [ { " " }, " ", Entities ] ;
EntityStart = [ dash ], nmstart, [ Block ] ;
Block = BlockNoDash | BlockNoUnderscore ;
BlockNoDash = (underscore, [BlockNoUnderscore]) | (nonunderscoredash, {nonunderscoredash},
[Block]) | (ElementSeparator, NonBlock) ;
BlockNoUnderscore = (dash, [BlockNoDash]) | (nonunderscoredash, {nonunderscoredash}, [Block])
| (ModifierSeparator, NonBlock) ;

NonBlock = NonBlockNoDash | NonBlockNoUnderscore ;
NonBlockNoDash = (nonunderscoredash, [NonBlock]) | (underscore, [NonBlockNoUnderscore]) ;
NonBlockNoUnderscore = (nonunderscoredash, [NonBlock]) | (dash, [NonBlockNoDash]) ;

ElementSeparator = underscore, underscore ;
ModifierSeparator = dash, dash ;
underscore = "_" ;
dash = "-" ;
nonunderscore = nonunderscoredash | dash ;
nondash = nonunderscoredash | underscore ;
nonunderscoredash = "a" | "b" | ... | "z" | "0" | "1" | ... | "9" ;
nmstart = nondash ;
nmchar = nonunderscore | underscore ;
```

Abbildung 19: EBNF für das B.E.M. Pattern

Um Platz zu sparen wurde beim Nichtterminalsymbol ***nonunderscoredash*** durch **…**angedeutet, dass hier noch mehr stehen müsste. Außerdem wurden Non-ASCII Zeichen und Unicodes nicht mit aufgenommen.

Die Schwierigkeit bei der Definition der Grammatik lag darin, dass es außer dem Whitespace zwischen einzelnen Entitäten keinen eindeutigen Trenner zwischen den einzelnen Token gibt. Die Zeichenfolge **__**, die einen Block-Bezeichner von einem Element-Bezeichner trennt, besteht aus einem Zeichen, das einzeln auch für einen Block-Bezeichner gültig ist, aber wenn es zweimal hintereinander auftaucht, ein anderes Token bildet. Analog gilt dies auch für die Zeichenfolge **--**, die einen Block Bezeichner von einem Modifier Bezeichner trennt.

Dies bedeutet, dass man sich durch die Einführung weiterer Produktionsregeln den Wert des letzten Zeichens merken muss, damit bestimmt werden kann, ob weiterhin ein Block gelesen wird oder durch ein weiteres Vorkommen des Zeichens ein Separator gelesen wurde.

3.4.1.2 **Spracheinbindung**

Das B.E.M. Pattern dient der Benennung von CSS Klassen in CSS Regeln (vgl. „B.E.M. Pattern“). Somit muss die B.E.M. Unterstützung an eben diese Klassennamen gebunden werden.

Wie im Kapitel 2.4.2.5 beschrieben unterstützt IntelliJ IDEA das Einfügen von Sprachen in die Elemente anderer Sprachen. Mit Hilfe dieses Features wird die im vorherigen Kapitel definierte B.E.M. Sprache eingebunden.

Um den Punkt der Injektion zu bestimmen habe ich CSS Dateien mit Hilfe des in IDEA integrierten ***PSI Context Viewer*** Tools analysiert, welchen man über den Menüeintrag ***Tools*** – ***View Psi Structure of Current File*** für den aktiven Editor öffnen kann.

Das Tool ist mit den Standard-Einstellungen von IDEA nicht sichtbar. Eine Möglichkeit, das Tool zu aktivieren, besteht darin, im ***bin***-Verzeichnis des IDEA Installations-Ordners die Datei ***idea.properties*** um einen Eintrag ***idea.is.internal=true*** zu ergänzen und die IDE neu zu starten.

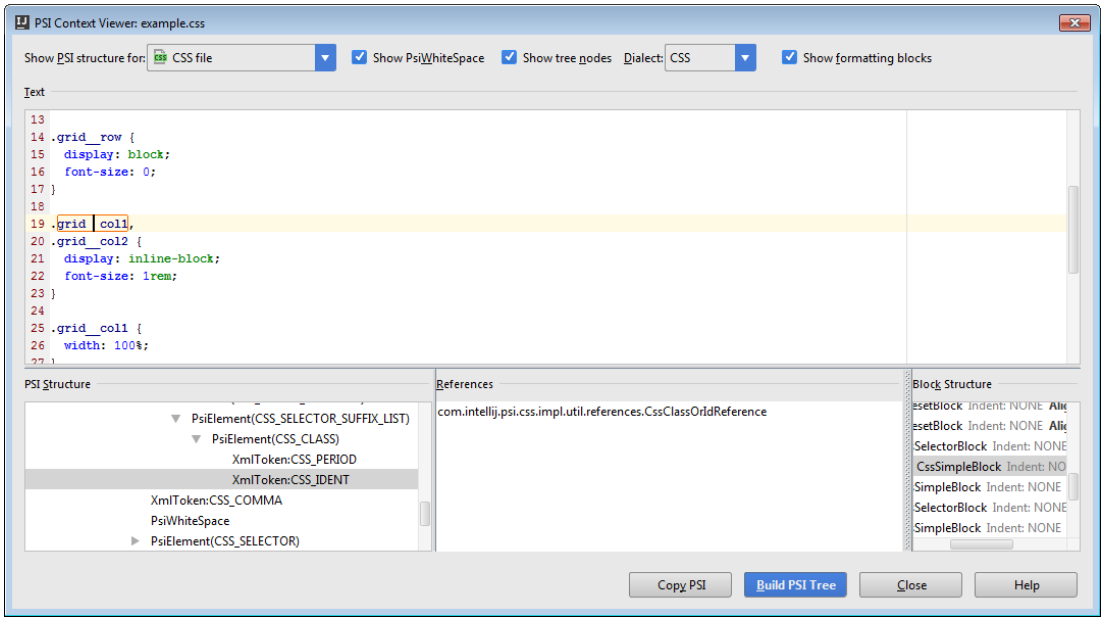


Abbildung 20: PSI Context Viewer

Abbildung 20 zeigt die Analyse des im Kapitel 3.3.1.3 gelieferten Beispiels mit Hilfe des PSI Context Viewers.

Die Ansicht wurde durch Verwendung des Menüeintrags ***Tool*** – ***View PSI Structure of Current File*** mit den aus dem aktiven Editor bekannten Daten initialisiert:

- Im oberen Bereich ist zu sehen, dass es sich um eine CSS Datei handelt und als ***Dialect*** ebenfalls ***CSS*** eingestellt ist, also keine spezielle Variante von CSS betrachtet werden soll.
- Mit Hilfe der Checkboxen lässt sich filtern, wie viele Details in den folgenden Bereichen dargestellt werden sollen.
- Der Bereich ***Text*** stellt den Inhalt des Editors dar aus dem die Ansicht erzeugt wurde.
- Unter ***Psi Structure*** wird ein aus dem ***Text*** Bereich generierter PSI-Baum dargestellt.
- Der Tooltip für die einzelnen Baum-Knoten zeigt die Java-Klasse an, die das PSI-Element implementiert.
- Die Bereiche ***References*** und ***Block Structure*** sind für die Analyse nicht weiter relevant, weil ein passendes PSI-Element gesucht werden soll.
- ***Copy PSI*** kopiert den in ***PSI Structure*** dargestellten Baum in das Clipboard.
- Wenn der Text verändert wurde, lässt sich mit Hilfe von ***Build PSI Tree*** der Inhalt von ***PSI Structure*** neu erzeugen.
- Eine Selektion in einem der Bereiche ***Text***, ***Psi Structure*** und ***Block Structure*** führt dazu, dass die Repräsentation in den jeweiligen anderen Bereichen auch entsprechend selektiert wird.

Durch Selektion eines CSS Klassennamen im Bereich „Text“ wurde ersichtlich, dass das PSI-Element mit der Implementationsklasse ***CssTokenImpl*** den eigentlichen Klassennamen repräsentiert. Da es sich aber bei näherer Betrachtung auch bei der CSS Eigenschaft ***display*** um ein ***CssTokenImpl*** handelt, ist der Kontext, in dem sich das Token befindet, relevant.

Der unmittelbare Elternknoten des in der oberen Abbildung zu sehenden Tokens ist ***CssClassImpl***. Dieser besteht neben dem ***CSS_IDENT*** Token, welches den Klassennamen repräsentiert, aus einem weiteren Token ***CSS_PERIOD***. Bei letzterem handelt es sich um den im Kapitel 2.1.3.2 beschriebenen Klassen-Selektor, der für die Namensgebung von B.E.M. nur so weit relevant ist, dass nach ihm ein Klassenname folgt.

Da sich durch den Namen und den Aufbau der ***CssClassImpl*** darauf schließen lässt, dass es sich hierbei um einen Klassen-Selektor handelt, sind alle geeigneten „Language Injection Hosts“ somit die Kind-Elemente des PSI-Elements ***CssClassImpl***, welche ein Token von Typ ***CSS_IDENT*** repräsentieren.

Da ***CssClassImpl*** die einzige Implementation des Interfaces ***CssClass*** darstellt, wird im Folgenden nur noch das Interface betrachtet.

Da der Host weder durch das Interface ***CssClass*** das Interface ***PsiLanguageInjectionHost*** erweitert, noch ***CssClassImpl*** als einzige Implementation das genannte Interface

implementiert, ist die Verwendung des Erweiterungspunkts ***languageInjectors*** ausgeschlossen.

Aus diesem Grunde muss wie in Kapitel 2.4.4.6 beschrieben eine eigene Implementation für das Interface bereitgestellt werden, damit der ***multiHostInjector*** verwendet werden kann.

3.4.2 Anwendung 1: Syntax-Highlighting für B.E.M.

Nach der Vorarbeit, die durch den Plugin-Kern geleistet wurde, beschränkt sich das Syntax-Highlighting auf die Verwendung des Erweiterungspunkts: ***syntaxHighlighterFactory*** (vgl. 2.4.4.3).

3.4.3 Anwendung 2: Farbeinstellungen für das B.E.M. Syntax-Highlighting

Die Color Settings Page baut auf der Implementierung von ***SyntaxHighlighter*** auf und wird den Erweiterungspunkt ***colorSettingsPage*** verwenden (vgl Kapitel 2.4.4.4).

3.4.4 Anwendung 3: B.E.M. Tool-Window

Das B.E.M Tool-Window wird mit Hilfe des Erweiterungspunkts ***toolWindow*** (vgl. Kapitel 2.4.4.7) eingebunden. Die einzelnen Komponenten des Tool-Windows werden mit Java Swing erzeugt.

Nach Analyse des Quellcodes ist ersichtlich, dass die StructureView für die Darstellung des Baums Klassen aus dem Package ***com.intellij.ide.util.treeView.smartTree*** verwendet, welche eine Vielzahl von Erweiterungen zum JTree der Java Swing API zur Verfügung stellt. Zu diesen Erweiterungen gehört unter anderem ein verbessertes Cache verhalten und „lazy loading“. [41] Daher soll dieses Package in der Implementierung verwendet werden.

3.4.5 Anwendung 4: Generierung von B.E.M. Entitäten

Für die Code-Generierung sollen zwei Mittel der API Anwendung finden: Zum einen die Verwendung von Actions (vgl. Kapitel 2.4.5) und zum anderen Live-Templates (vgl. (JetBrains Documentation Team 2016)). Die Live-Templates sind hierbei nicht als feste Datei hinterlegt, sondern werden von den Actions dynamisch generiert, sodass ihre Vorbelegungen beeinflusst werden können.

[41] (JetBrains Development Team 2016)

4 Implementierung und Test

Die folgenden Kapitel beschreiben die Implementierung und Tests für das B.E.M. Plugin. Ich verweise hiermit auf den beigelegten Quellcode und werde nur die Code-Stellen näher erläutern, bei denen es bei der Implementierung Besonderheiten gab. Große Teile des Codes lassen sich anhand seiner Code-Dokumentation bereits insofern nachvollziehen, dass weitere Erläuterungen nur zu Redundanz führen würden.

Bei der Code-Dokumentation sei angemerkt, dass diese auf Englisch verfasst wurde.

4.1 Benutzungsoberfläche

IntelliJ IDEA gibt einen großen Teil der Benutzungsoberfläche bereits fest vor, darum gibt es in diesem Kapitel lediglich eine Beschreibung für die Benutzungsoberfläche, die für User Story 3 erstellt wird (vgl. 3.3.3). Auch Funktionalitäten, die die Code-Analyse betreffen und keinen eigenen Dialog anzeigen oder wo der Dialog bereits von IntelliJ bereitgestellt wird, werden unter dem Aspekt der Benutzungsoberfläche nicht weiter betrachtet.

Da das B.E.M. Tool-Window ähnlich aussehen soll, wird zunächst die integrierte Structure View von IDEA beschrieben.

4.1.1 Structure View

Die Structure View wird eingeblendet, indem man im Menü unter ***View*** – ***Tool Windows*** den Eintrag ***Structure*** auswählt.

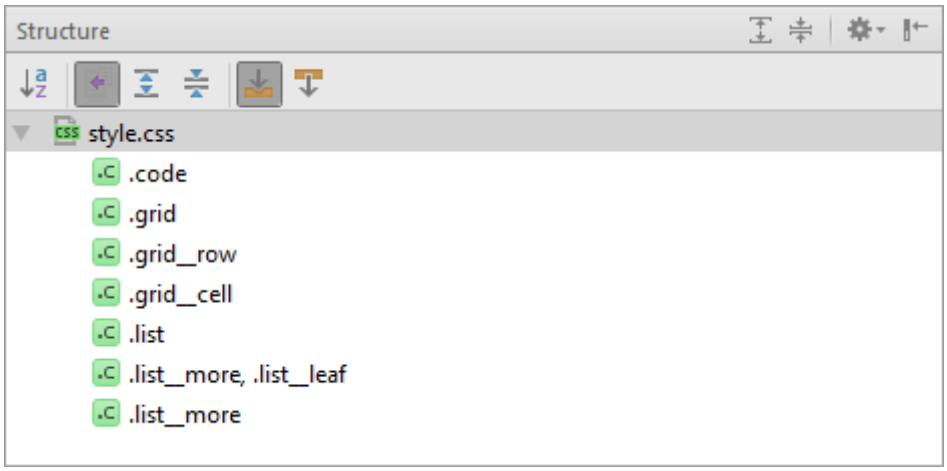


Abbildung 21: „Structure View" einer CSS Datei

Wie in Abbildung 21 zu sehen besteht die Ansicht aus drei Bereichen:

- Die Kopfzeile definiert den Namen der Ansicht und Kontrollelemente zur Steuerung der Ansicht
- Die Toolbar der ***Tree Control*** Komponente enthält Kontrollelemente zur Steuerung und Filtern der Ansicht.
- Die ***Tree Control*** Komponente zeigt die interne Struktur der CSS Datei als Baum an. Hierbei orientiert sich die Komponente an der Struktur des PSI Modells, welches beim Auslesen der CSS Datei erzeugt wurde.

  Ohne weitere Anpassungen wird die CSS-Datei als Wurzel-Knoten genommen und dann ein Baum erstellt, der Regel-Mengen oder Import-Anweisungen als Blatt-Knoten hat. Zwischen Wurzel und Blattknoten können sich zudem weitere Elemente befinden, sofern sich die Regelmenge in einer at-Regel, wie z.B. einer Media Query befindet (vgl. Kapitel 2.1.3.1).

4.1.2 B.E.M. Tool-Window

Wie bereits in User Story 3: B.E.M. Tool-Window beschrieben, soll das Aussehen, des Fensters an das Aussehen der Structure View angelehnt sein, welche im vorherigen Kapitel beschrieben wurde. Dies bestimmt bereits eindeutig, wo sich die Toolbar und deren Funktionalitäten befinden sollen.

Der Aspekt, der neu ist, ist der Inhalt der ***Tree Control***, welcher nun die aggregierten B.E.M. Entitäten der aktuellen Datei enthalten soll. Da es sich um eine Baum Struktur handelt, gibt es mehrere Möglichkeiten, die Entitäten darzustellen.

Hierzu habe ich im Vorfeld einen Prototyp in HTML modelliert, der die B.E.M. Aggregation aus dem Code von Abbildung 17 darstellen sollte. Hierbei wird auf die Darstellung des Dateinamens als Root-Knoten verzichtet, da dieser immer gleich ist.

Es folgen die Phasen bis zum finalen Entwurf für die Benutzungsoberfläche der ***Tree Control***. Die Icons, die die Buchstaben ***B***, ***E*** und ***M*** abbilden kommen aus dem für nicht kommerzielle Zwecke zur Verfügung gestellten Icon-Paket „Multipurpose Alphabet“[42].

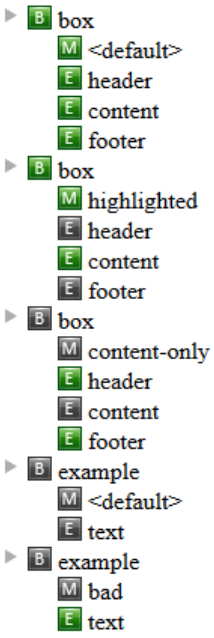


Abbildung 22: B.E.M. Tree (Prototyp 1)

Die Idee hinter dieser Darstellung war es, für alle Modifier eines Blocks einen eigenen Knoten im Baum anzulegen.
Der erste Knoten war hierbei immer der Block ohne Modifier.

Anforderungen:

- ✓ Unterscheidung der Typen (Block, Element, Modifier)
- ✓ Darstellung aller verwendeten B.E.M. Entitäten (grün)
- ✓ Darstellung der impliziten B.E.M. Entitäten (grau)

Nachteile:

- Man sieht erst nach dem Aufklappen, um welchen Modifier es sich handelt
- Es gibt viele Elemente auf Ebene 1, was bei vielen B.E.M. Entitäten sehr unübersichtlich werden kann
- Informationsredundanz: Das Icon auf Ebene 1 liefert dieselbe Information wie ihr jeweils erstes Icon auf Ebene 2 (z.B. geben box und <default> beide an, dass die Entität verwendet wird).

[42] (Nayak 2008)

In der 2. Version habe ich die Redundanz entfernt und die Informationsredundanz entfernt, indem ich den Modifier Bezeichner mit in Ebene 1 übertragen habe.

Anforderungen:

- ✓ Unterscheidung der Typen (Block, Element, Modifier)
- ✓ Darstellung aller verwendeten B.E.M. Entitäten (grün)
- ✓ Darstellung der impliziten B.E.M. Entitäten (grau)

Nachteile:

- ~~Man sieht erst nach dem Aufklappen, um welchen Modifier es sich handelt~~
- Es gibt viele Elemente auf Ebene 1, was bei vielen B.E.M. Entitäten sehr unübersichtlich werden kann.
- ~~Informationsredundanz: Das Icon auf Ebene 1 liefert dieselbe Information wie ihr jeweils erstes Icon auf Ebene 2 (z.B. geben box und <default> beide an, dass die Entität verwendet wird).~~

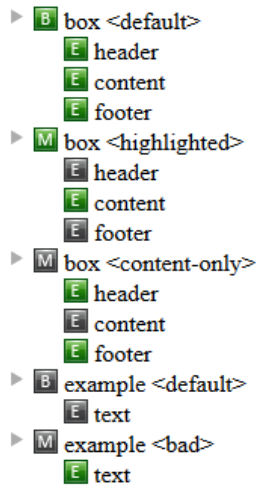


Abbildung 23:
B.E.M. Tree (Prototyp 2)

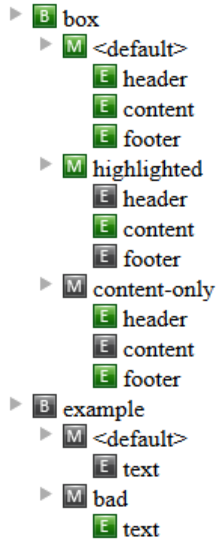


Abbildung 24:
B.E.M. Tree (Prototyp 3)

Die nächste Version zielte darauf ab, die Anzahl der Knoten, die sich auf Ebene 1 befinden, zu reduzieren. Hierfür wurden alle Knoten, die zu demselben Block gehören in eine weitere Ebene umgezogen, sodass nur noch ein Knoten übrig war, der den Block Bezeichner repräsentiert.

Anforderungen:

- ✓ Unterscheidung der Typen (Block, Element, Modifier)
- ✓ Darstellung aller verwendeten B.E.M. Entitäten (grün)
- ✓ Darstellung der impliziten B.E.M. Entitäten (grau)

Nachteile:

- ~~Es gibt viele Elemente auf Ebene 1, was bei vielen B.E.M. Entitäten sehr unübersichtlich werden kann.~~
- Um die Elemente eines Blocks zu sehen muss man nun Knoten ausklappen.
- Insgesamt sind mehr Klicks erforderlich, um die Aggregation für einen Block zu betrachten.
- Regression: Die Informationsredundanz ist für den <default> Modifier wieder da.

In der letzten Version habe ich alle normalen Elemente wieder in Ebene 2 umgezogen, und damit die Informationsredundanz beseitigt.

Anforderungen:

- ✓ Unterscheidung der Typen (Block, Element, Modifier)
  Verbesserung: Modifizierte Elemente lassen sich durch die unterschiedlichen Ebenen klarer von normalen Elementen unterscheiden, ohne ein neues Icon einführen zu müssen.
- ✓ Darstellung aller verwendeten B.E.M. Entitäten (grün)
- ✓ Darstellung der impliziten B.E.M. Entitäten (grau)

Nachteile:

- ~~Um die Elemente eines Blocks zu sehen muss man nun Knoten expanden.~~
- Insgesamt sind mehr Klicks erforderlich, um die

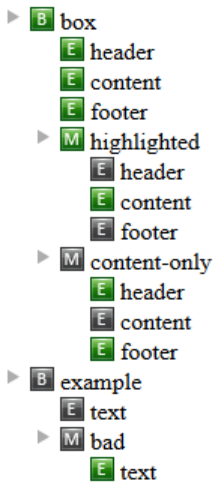


Abbildung 25:
B.E.M. Tree (Prototyp 4)

Aggregation für einen Block zu betrachten.

- ~~Regression: Die Informationsredundanz ist für den <default> Modifier wieder da.~~

Für den letzten verbleibenden Nachteil wäre es wieder nötig zu geworden, die zusätzliche Ebene aufzulösen und mehr auf Ebene 1 zu verlagern, was dann wieder zu Prototyp 2 geführt hätte.

Aus diesem Grund habe die Vor- und Nachteile von Prototyp 2 und Prototyp 4 abgewogen, was letztendlich zu folgender Fragestellung geführt hat:

Soll die Ansicht eher eine schnellere Informationsgewinnung durch weniger Klicks bieten, dafür aber bei vielen B.E.M. Entitäten weniger übersichtlich sein oder soll eine größere Übersichtlichkeit durch striktere Kategorisierung in Form von Ebenen zur Verfügung stellen, welche aber einen Klick mehr erfordert, um an alle Informationen zu kommen?

Das B.E.M. Pattern wurde - wie bereits in den Grundlagen erwähnt – für größere Projekte entwickelt, um Frontend Komponenten besser in kleinere Module aufteilen zu können.
In solchen Projekten, wird es also sehr viele B.E.M. Entitäten geben, was meiner Ansicht nach die Problematik, die Prototyp 2 verursacht sehr verstärkt während der zusätzliche Klick auch durch Verwendung der ***Expand All*** Funktionalität verhindert werden kann.

Die Implementierung wird daher Prototyp 4 in das B.E.M. Tool-Window übertragen.

4.2 Programmierung und Architektur

Die folgenden Kapitel befassen sich mit der Implementierung der einzelnen User-Stories und stellen die Architektur von Modellen vor, die unabhängig von der Architektur der Erweiterungspunkte benötigt werden.

4.2.1 Einrichtung der Entwicklungsumgebung

IntelliJ IDEA Plugins werden in der IntelliJ IDEA Entwicklungsumgebung geschrieben (vgl. Kapitel 2.4.3). Hierzu gibt es den Projekt-Typ ***IntelliJ Platform Plugin***. Bevor ein neues Projekt erstellt werden kann, müssen zunächst eine spezielle ***JDK*** Konfiguration und ein ***IntelliJ Platform Plugin SDK*** definiert werden.

Zur Einrichtung des JDK muss zunächst der ***Project Structure*** Dialog geöffnet werden (***File*** - ***Project Structure***). In diesem findet man unter ***Platform Settings*** - ***SDKs*** einen ***Add New SDK*** Button mit dessen Hilfe man ein neues SDK hinzufügen kann.

4.2.1.1 IDEA JDK

Voraussetzung für die Plugin Entwicklung in IDEA 15 das JDK 8, welches auf dem System installiert werden muss. Dieses findet sich auf der Oracle Homepage unter Downloads. Für diese Ausarbeitung wird die Version 1.8.0 mit der Build-Nummer 74 verwendet (vgl. Anhang 3).

Das installierte JDK muss nun als neues SDK vom Typen ***JDK*** hinzugefügt werden. Als ***Home Folder*** muss das Installationsverzeichnis ausgewählt werden. Nach der Erstellung muss der mit ***1.8*** vorbelegte Name nun exakt in ***IDEA jdk*** umbenannt werden, da es sich hier um einen festgelegten Bezeichner handelt.

Was nun das benutzerdefinierte JDK von anderen JDKs unterscheidet, ist die Ergänzung der jar-Datei ***lib/tools.jar*** aus dem JDK 8 Home Folder, welche zusätzlich zu den anderen Pfaden in den Classpath des JDKs eingetragen werden muss.

4.2.1.2 IntelliJ Platform Plugin SDK

Zur Unterstützung des Debuggings bei der Plugin Entwicklung empfiehlt JetBrains, den Source-Code der „IntelliJ IDEA Community Edition“ in der zur Installation passenden Version herunterzuladen.
Dies erfolgt über die Git-URL: ***git://git.jetbrains.org/idea/community.git*** und wird hier nicht im Detail erläutert.

Nun muss analog zum vorherigen Kapitel ein ***IntelliJ Platform Plugin SDK*** hinzugefügt werden. Hierzu wird das Installations-Verzeichnis von IntelliJ IDEA 15 als ***Home Folder*** ausgewählt. Im nachfolgenden Schritt wählt man das zuvor erstellte ***IDEA jdk*** als ***Java SDK***.

Für die zuvor erwähnte Debugging Unterstützung fügt man nun den Pfad, an dem der Source-Code der „IntelliJ IDEA Community Edition“ liegt zu den Pfaden im Tab ***Sourcepath*** hinzu.

4.2.1.3 Einrichtung des Projekts

Nachdem die SDKs angelegt sind kann das Projekt mit Hilfe des Menüeintrags ***File*** – ***New*** – ***Project*** erzeugt werden. Hierzu wird der Typ ***IntelliJ Platform Plugin*** ausgewählt. Als ***Project SDK*** wird das im vorherigen Kapitel erstellte ***IntelliJ Platform Plugin SDK*** ausgewählt.

IDEA wird nun ein Grundgerüst für das Projekt erstellen. In diesem Grundgerüst befindet sich unter anderem im Ordner ***META-INF*** die Datei ***plugin.xml***, die das IDEA Plugin beschreibt. Nach Eingabe der Meta Informationen des Plugins ist es wichtig, zwei Abhängigkeiten hinzuzufügen:

```
<depends>com.intellij.modules.lang</depends>
<depends>com.intellij.css</depends>
```

Die Abhängigkeit zu ***com.intellij.modules.lang*** aktiviert die entsprechenden Erweiterungen für Sprachen und die Abhängigkeit zu ***com.intellij.css*** sorgt für den Zugriff auf die CSS Sprache, welche erst durch ein Plugin hinzugefügt wird.

Damit die zweite Abhängigkeit zum ***com.intellij.css*** Package erkannt wird, muss man die entsprechende Bibliothek noch hinzufügen. Dies erfolgt mit Hilfe des Dialogs ***Project Structure***, der über den gleich benannten Menüeintrag unter ***File*** erreichbar ist. Hier findet man unter ***Project Settings*** – ***Modules*** im rechten Tab den Eintrag ***Dependencies***. Durch die Verwendung des ***Add*** Buttons wird eine Abhängigkeit vom Typ ***JARs or directories*** hinzugefügt. Die hinzuzufügende Datei liegt im IntelliJ IDEA 15 Home Folder unter ***plugins/CSS/lib/css.jar***. Nach dem Hinzufügen muss in der Spalte ***Scope*** noch ***Provided*** ausgewählt werden, was dem Compiler sagt, dass die Abhängigkeit bereits durch ein SDK oder durch den Container (also in diesem Fall IDEA) zur Laufzeit hinzugefügt wird. Zudem sollte die Reihenfolge der Abhängigkeiten so angeordnet werden, dass zuerst das ***IntelliJ Platform Plugin SDK***, dann die ***css.jar*** und dann der Eintrag ***<Module source>*** in der Liste stehen.

4.2.2 Sprachdefinition

Dieses Kapitel implementiert die in Kapitel 3.4.1.1 beschriebene Sprache.

Wie in Kapitel 2.4.4.2 beschrieben, erwartet das Interface ***ParserDefinition*** auch einen Lexer. Der Lexer trennt einen String in Tokens auf, auf die der Parser Zugriff hat und mit deren Hilfe er PSI-Element erzeugen kann. Die Aufteilung der Tokens bestimmt zudem die Granularität auf der das Syntax-Highlighting erfolgen kann, da das Interface ***SyntaxHighlighting*** die Angabe eines Lexer erwartet. Da Kapitel 3.3.1.1 festlegt, welche Teile einer B.E.M. Entity unterschiedlich dargestellt werden sollen, bestimmt das Syntax-Highlighting bereits, welche Token der Lexer generieren muss.

4.2.2.1 Lexer

IntelliJ IDEA hat - wie in der SDK Dokumentation[43] beschrieben - eine angepasste Version von JFlex, die dazu verwendet werden kann, Lexer zu erzeugen, die kompatibel zum eingebauten ***FlexAdapter*** sind. Hierzu muss das ***Grammar-Kit*** Plugin installiert sein.

[43] (JetBrains Documentation Team 2016)

Da es für diese Integration des Lexers von JetBrains eine entsprechende Dokumentation gibt und die zugrunde liegende Definition des IDENT Tokens bereits in Flex-Notation vorliegt, beschreibe ich die benötigten B.E.M. Tokens in der in Flex Notation. Diese befindet sich in Anhang 2.

Die Richtigkeit des Lexers wird im Kapitel 4.3 sichergestellt. Eine Optimierung hinsichtlich der Performance wird bereits durch JFlex durchgeführt (Klein, Rowe und Décamps 2015), sodass ich mehr Wert auf die Lesbarkeit der Flex-Notation gelegt habe.

Leider konnten von der EBNF Variante nur wenig übernommen werden. Hier war das Problem, dass die Produktionsregeln nicht so zerlegt werden konnten, dass die einzelnen Token auch alleine hätten geparst werden können. Dies wäre die Voraussetzung dafür gewesen, dass man die EBNF in Form von Makros in der Flex-Notation hätte wiederverwenden können.

Da in der Flex-Notation im gleichen State beim Zutreffen mehrerer Regeln die maximale Größe des konsumierten (Teil-)Tokens und nicht die Definitionsreihenfolge Vorrang hat (vgl. (Klein, Rowe und Décamps 2015)), musste dafür gesorgt werden, dass die Regeln, die die Identifier auslesen zunächst nur ein Zeichen lesen und ein großer „look ahead“ für die Regeln gemacht wird, die für einen Übergang in das Auslesen der anderen Entitäten sorgen.

Dafür konnte jedoch dann die bereits in Abbildung 18 gezeigte Definition einbezogen werden. Hier mussten lediglich alle ASCII Charakter, die in einem Makro zulässig waren, um ihre groß geschriebene Variante erweitert werden. Zwar bietet JFlex ein Flag ***%ignorecase***, aber dieses greift nicht für Makros (vgl. (Klein, Rowe und Décamps 2015)).

Der Lexer generiert also die folgenden Token:

- **BemTypes.BLOCK_IDENTIFIER**
  Repräsentiert den Bezeichner eines Blocks

- **BemTypes.ELEMENT_IDENTIFIER**
  Repräsentiert den Bezeichner eines Elements

- **BemTypes.MODIFIER_IDENTIFIER**
  Repräsentiert den Bezeichner eines Modifiers

- **BemTypes.ELEMENT_SEPARATOR**
  Stellt den Trenner zwischen Block Bezeichner und Element Bezeichner dar

- **BemTypes.MODIFIER_SEPARATOR**
  Stellt den Trenner zwischen Block Bezeichner und Modifier Bezeichner dar

- **BemTypes.NO_BEM**
  Stellt ein Token, welches nicht dem B.E.M. Pattern entspricht, dar.

- **TokenType.WHITE_SPACE**
  Vordefinierter Tokentyp, der jegliche Art von Whitespaces darstellt

- **TokenType.BAD_CHARACTER**
  Vordefinierter Tokentyp, der ein unerwartetes Zeichen darstellt

4.2.2.2 Parser

Aus den im vorherigen Kapitel gezeigten Tokens muss der Parser nun eine Beschreibung für die Erzeugung der einzelnen PSI-Elemente angeben. Hierfür bietet das bereits erwähnte ***Grammar-Kit*** Plugin die Möglichkeit aus einer BNF-Datei die Klassenstruktur zu generieren, die der in der Datei in EBNF angegebenen PSI-Struktur entspricht.

```
bemFile ::= entity_*
private entity_ ::= (element|modifier|block|NO_BEM|CRLF)
element ::= (BLOCK_IDENTIFIER ELEMENT_SEPARATOR ELEMENT_IDENTIFIER)
modifier ::= (BLOCK_IDENTIFIER MODIFIER_SEPARATOR MODIFIER_IDENTIFIER)
block ::= (BLOCK_IDENTIFIER)
```

Abbildung 26: EBNF für den B.E.M. Parser

Die grundlegende EBNF in der für das Plugin benötigten Form, welche aus den Tokens einzelne B.E.M. Entitäten erzeugt, zeigt Abbildung 26. Während die Syntax große Ähnlichkeit mit der in Kapitel 2.5 gezeigten Syntax aufweist, wird sie durch entsprechende Schlüsselwörter und Konstrukte angereichert. So bedeutet das oben gezeigte Schlüsselwort ***private***, dass die Produktionsregel keine eigene Klasse abbilden soll.

Der beigelegte Quellcode des B.E.M. Plugins enthält die vollständige Definition der BNF-Datei. Hier gilt es zu erwähnen, dass ich jede generierte B.E.M. Entität durch Ableitung der Klasse ***BemEntityImpl*** das Interface ***BemEntity*** implementieren lasse. Hierdurch ist es möglich, auf allen Entitäten die Methode ***getBlockIdentifier*** aufzurufen, welche den Block Bezeichner, den jede Entität beinhaltet (vgl. Kapitel 2.3.1), zurückgibt.

4.2.2.3 Klassenstruktur

Durch die Generierung des Parser-Codes werden Klassen und Interfaces für PSI-Elemente erzeugt. Die Relation der einzelnen Klassen und Diagramme zeigt Abbildung 27.

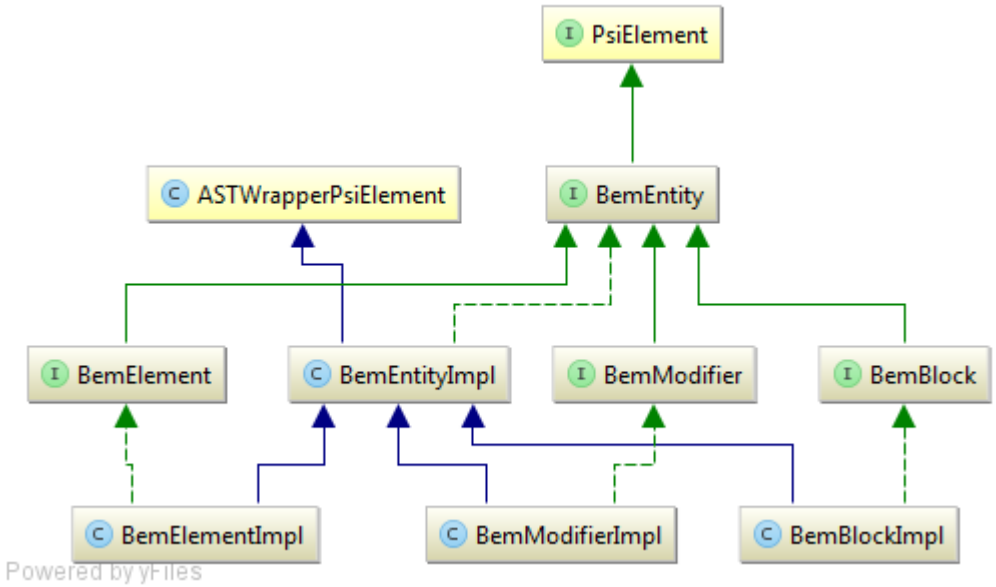


Abbildung 27: Klassendiagramm der PSI-Elemente des B.E.M. Plugins

Hinsichtlich der Nutzung des Erweiterungspunkts ***lang.parserDefinition*** ergibt sich das in der nächsten Abbildung gezeigte Diagramm ausgehend von der Implementierungsklasse ***BemParserDefinition***.

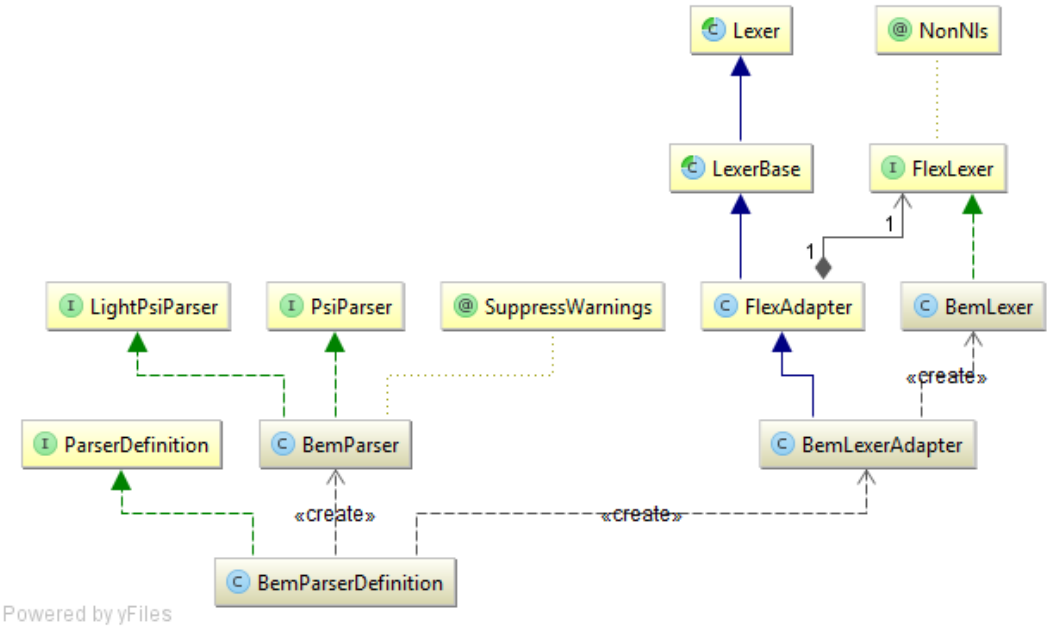


Abbildung 28: Klassendiagramm der Parser-Definition

4.2.3 Language Injection

Wie in Kapitel 3.4.1.2 bereits analysiert, muss die B.E.M. Sprache in das PSI-Element eingefügt werden, welches das Token ***CSS_IDENT*** repräsentiert und sich innerhalb eines PSI-Elements befindet, welches ***CssClass*** implementiert. Dieser wird im Folgenden als „CSS Host“ bezeichnet.

Kapitel 2.4.4.6 hat gezeigt, dass ***PsiLanguageInjectionHost*** das Interface ***PsiElement*** erweitert, welches der „CSS Host“ bereits implementiert. Die Teilmenge der zu implementierenden Methoden, die sich aus dem genannten Interface ergibt, konnte also durch Delegation an den „CSS Host“ implementiert werden, wodurch nur noch die Methoden übrig geblieben sind, die das Interface ***PsiLanguageInjectionHost*** hinzufügt.

Da es sich bei der Implementierung eines Hosts durch Delegation um eine Möglichkeit handelt, die nicht nur auf die Sprachinjektion des B.E.M. Plugins beschränkt ist, habe ich eine abstrakte Klasse ***AbstractDelegatingInjectionHost*** erstellt, welche die Delegation bereits für beliebige PSI-Elemente vornehmen kann, und im Paket ***com.holtkamp.intellij.common.lang*** liegt.

Im Folgenden wird die Implementation der übrigen Methoden beschrieben:

- **isValidHost**
  Das Ergebnis dieses Aufrufs gibt an, ob die aktuelle Instanz Injektions akzeptiert. Da dies immer möglich sein soll, wird ***true*** zurückgegeben.

- **updateText**
  Laut JavaDoc der Methode soll hier an die Methode ***handleContentChange*** der Klasse ***ElementManipulators*** delegiert werden, sofern eine Implementation von ***ElementManipulator*** für das Host Element registriert ist. Das Host Element ist vom Typ ***XmlToken*** und eine Betrachtung der Implementationen von ***ElementManipulator*** hat ergeben, dass es eine Implementation für ***XmlToken*** gibt, wodurch die oben beschriebene Delegation möglich war.

- **createLiteralTextEscaper**
  Der ***LiteralTextEscaper*** wird benötigt, um den Inhalt des „CSS Host“ in den Inhalt der B.E.M. Datei zu konvertieren. Da an der Repräsentation nichts verändert werden musste, war der ***StringLiteralEscaper*** die passende Wahl, weil nach Betrachtung des Quellcodes der gesamte String unverändert an die injizierte Sprache übertragen wird.

Da auch hier eine allgemeine Implementation für Tokens, deren Inhalt bei Nutzung in einer injizierten Sprache nicht weiter verändert werden soll, vorliegt, habe ich auch diese

Implementierung in das Paket ***com.holtkamp.intellij.common.lang*** verschoben und eine allgemeinere Klasse ***TokenInjectionHost*** daraus erstellt.

Um den für Sprachinjektion typischen grünlichen Hintergrund im Editor-Fenster zu unterdrücken, muss das Interface ***InjectionBackgroundSuppressor*** implementiert werden, welches keine weitere Methoden erfordert sondern lediglich als Auszeichnung des Hosts dient.

Das aus der Implementierung der Sprachinjektion resultierende Klassendiagramm zeigt Abbildung 29:

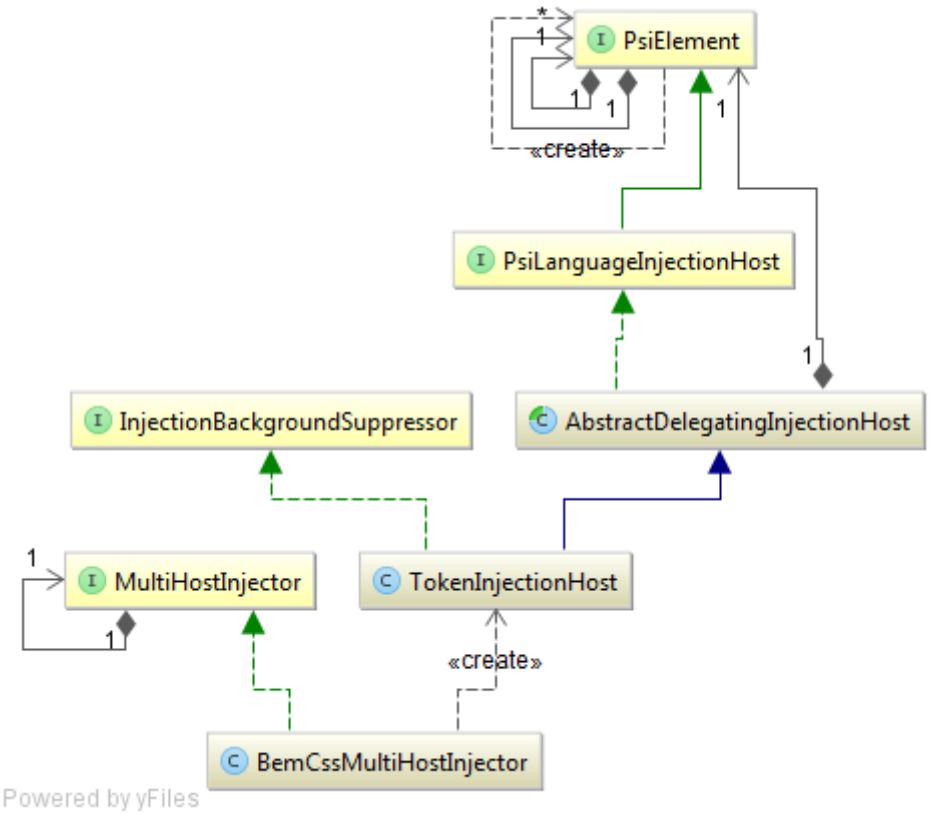


Abbildung 29: Klassendiagramm der Sprachinjektion

Leider gab es ein unerwartetes Problem nach der Sprachinjektion: Das Feature ***Find Usages*** war für CSS-Klassen nicht mehr aus dem Editor benutzbar. Das PSI-Element, welches ***CssClass*** implementiert, wurde vollständig von der jeweiligen ***BemEntity*** Implementierung verdeckt. Um das Feature wiederherzustellen musste der Erweiterungspunkt ***lang.findUsagesProvider*** für die Sprache B.E.M. implementiert werden. Hierzu wurde in der Plugin-Definition der ***CssFindUsagesHandler*** als ***implementationClass*** verwendet.

Nähere Informationen zum Erweiterungspunkt ***lang.findUsagesProvider*** stellt (JetBrains Documentation Team 2016) bereit.

4.2.4 Syntax-Highlighting für B.E.M.

Aufbauend auf der Sprachdefinition und -einbindung der vorherigen Kapitel, ließ sich das Syntax-Highlighting durch Verwendung des Erweiterungspunkts ***lang.syntaxHighlighterFactory*** implementieren.

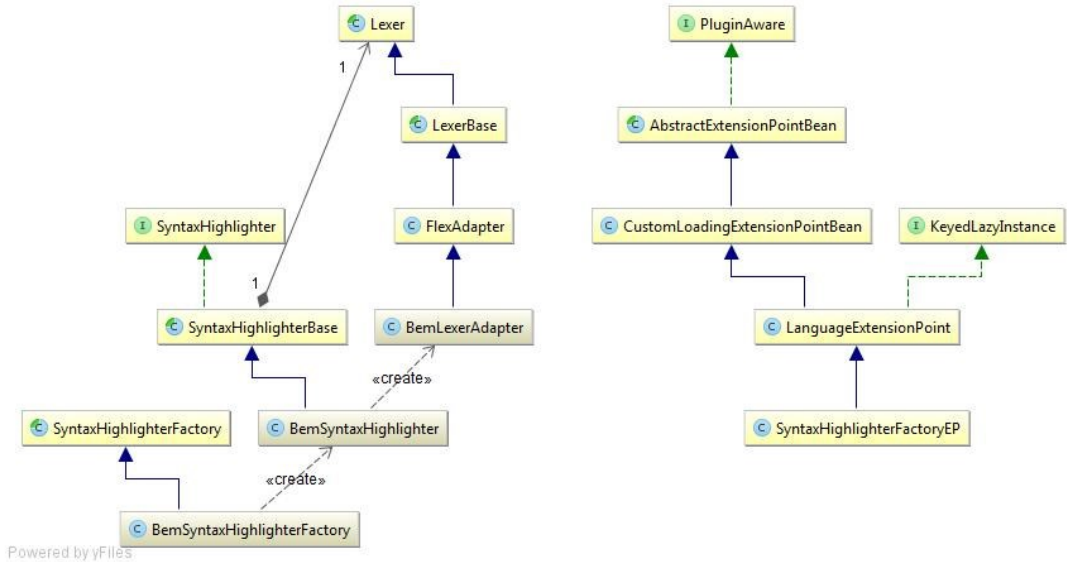


Abbildung 30: UML-Diagramm des B.E.M. Syntax-Highlightings

Wie in der oberen Abbildung zu sehen, erzeugt die Klasse ***BemSyntaxHighlighterFactory*** einen neuen ***SyntaxHighlighter*** vom Typ ***BemSyntaxHighlighter***, welcher wiederum einen ***BemLexerAdapter*** herstellt. Die ***plugin.xml***, die nicht im Diagramm aufgeführt ist, stellt dann die Verbindung mit der Klasse ***SyntaxHighlighterFactoryEP*** her.

Der ***BemSyntaxHighlighter*** stellt die Kernlogik für das Syntax-Highlighting bereit, indem er bestimmten Implementierungen von ***IElementType*** eine Liste von ***TextAttributesKey*** zuweist.

4.2.5 Color Settings Page

Für die Einbindung einer „Color Settings Page“ für das B.E.M. Plugin habe ich den entsprechenden Erweiterungspunkt ***colorSettingsPage*** verwendet (vgl. Kapitel 2.4.4.4).

Dies ließ sich analog zu dem in der Dokumentation [44] angegebenen Beispiel implementieren.

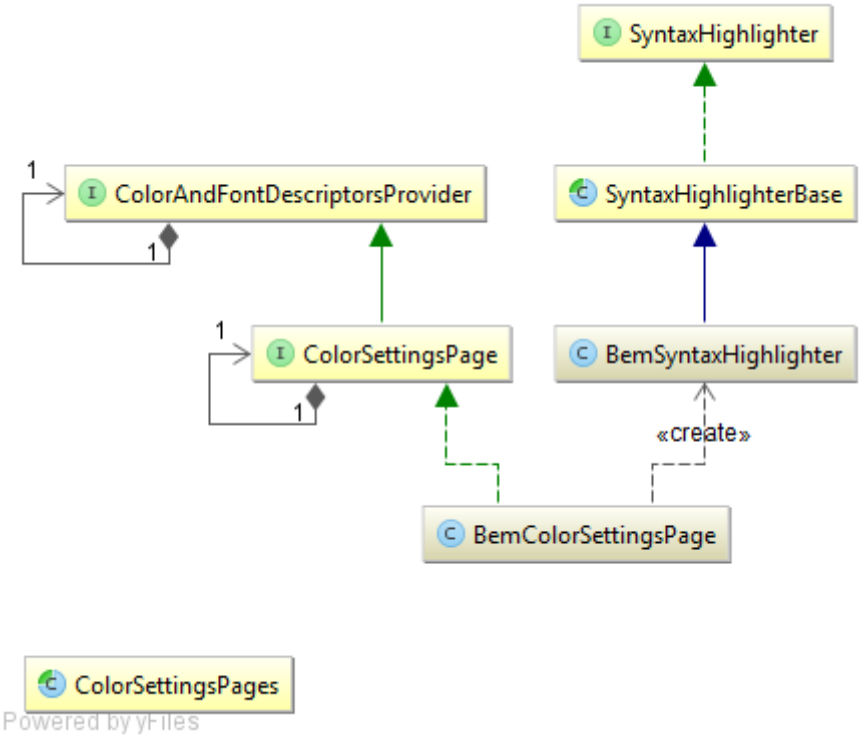


Abbildung 31: UML-Diagramm der B.E.M. Color Settings Page

Die ***plugin.xml*** sorgt durch Verwendung des Erweiterungspunkts ***colorSettingsPage*** dafür, dass die ***BemColorSettingsPage*** in der Klasse ***ColorSettingsPages*** registriert wird (vgl. obere Abbildung).

Das durch den Rückgabewert der Methode ***getDemoText*** definierte Code-Beispiel wurde so auf B.E.M. angepasst, dass alle vom ***BemSyntaxHighlighter*** angepassten Einstellungen, die sich auf Implementationen von ***IElementType*** beziehen, sichtbar sind.

In Anhang 5 ist zu erkennen, wie die B.E.M. Color Settings Page in IDEA dargestellt wird. Auch wird das zuvor erwähnte Code-Beispiel angezeigt und entsprechend der aktuellen Belegung eingefärbt.

[44] (JetBrains Documentation Team 2016)

4.2.6 B.E.M. Tool-Window

Die Implementation des in den Kapiteln 3.3.3 und 3.4.4 beschriebenen B.E.M. Tool-Windows ist zweigeteilt: den ersten Teil bildet das Befüllen der Ansicht mit Hilfe der Analyse des PSI-Baums, den die im aktuellen Editor geöffnete CSS-Datei bereitstellt, und den zweiten Teil die Erstellung der Ansicht mit Java Swing.

4.2.6.1 Extraktion der B.E.M. Informationen aus dem PSI-Baum

Würde man alle PSI-Elemente, die B.E.M. Entitäten repräsentieren, aus einer Datei auslesen und in eine Liste packen, so wäre nicht nur das Risiko von redundanten Informationen vorhanden sondern es entfällt auch die Relation der Entitäten zueinander, die sich durch ihre Reihenfolge innerhalb eines Selektors ergeben.

Sie würden zum Beispiel die in Kapitel 3.3.3.1 definierten impliziten Entitäten nicht mehr richtig ermittelt werden können. Auch ein modifiziertes Element ließe sich nicht mehr von einem normalen Element unterscheiden.

Um B.E.M. Informationen strukturiert darzustellen, habe ich daher den ***BemAggregator*** eingeführt. Dieser bildet einzelne B.E.M. Entitäten als Baum-Struktur ab und arbeitet auf Selektoren, um alle relevanten Informationen zu erhalten.

Jeder Block, jedes Element und jeder Modifier ist hierbei innerhalb der aktuellen Hierarchie einzigartig. Mehrere Ausprägungen werden zu einer Entität zusammengefasst, wobei die jeweiligen Referenzen auf PSI-Elemente gespeichert werden.

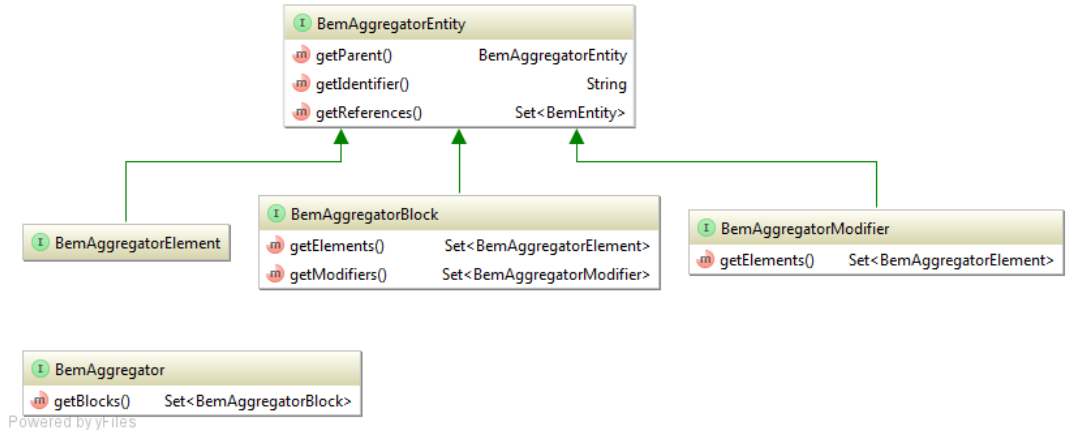


Abbildung 32: Klassendiagramm des B.E.M. Aggregators

Wie in der oberen Abbildung zu sehen ist, bietet der ***BemAggregator*** zunächst Zugriff auf seine Blocks, die einzelnen Instanzen von ***BemAggregatorBlock*** wiederum erlauben einen Zugriff auf deren Elemente und Modifier. Während Instanzen von ***BemAggregatorElement*** immer ein Blatt des Baums repräsentieren, bieten Instanzen von ***BemAggregatorModifier*** wiederum Zugriff auf deren Instanzen von ***BemAggregatorElement***. Diese bilden die modifizierten Elemente ab.

Da jede konkrete Ausprägung außer der ***BemAggregator*** von ***BemAggregatorEntity*** erbt, ist ein Zugriff auf die PSI-Elemente, die das Interface ***BemEntity*** (vgl. Kapitel 4.2.2.2) implementieren möglich. Ist die hier gelieferte Menge leer, so handelt es sich um eine implizite B.E.M. Entität.

Da der Baum nur an Hand der aus den Selektor direkt ermittelbaren B.E.M. Entitäten erzeugt wird, gibt es in der Utility-Klasse zwei Methoden, die alle impliziten und expliziten B.E.M. Elemente zurückliefert. ***getElementsAndModifiedElements*** macht dies hierbei für einen gegebenen ***BemAggregatorBlock***, während ***getElementsAndInheritedElements*** dies an Hand eines ***BemAggregatorModifier***s durchführt.

Mit Hilfe dieser Informationen lässt sich der B.E.M. Baum für das Tool-Window erzeugen.

4.2.6.2 Erstellung der Benutzungsoberfläche mit Java Swing

Für die Erstellung des Tool-Windows mit Java Swing war der Quellcode der vorhandenen Implementation der ***StructureViewComponent*** sehr hilfreich, da man eine Menge des Codes wiederverwenden konnte, solange dieser nicht zu sehr an die einzelnen Klassen der Structure View gekoppelt war.

Auffällig war hierbei eine unnötige enge Kopplung, die sich für die richtige Nutzung des Interface ***TreeActionsOwner*** und dessen Hilfsklassen ergeben hätte. Während das Interface selbst nichts über die konkrete Implementierung weiß, geht die Hilfsklasse ***TreeModelWrapper*** davon aus, dass das zu umschließende Model eine Implementierung von ***StructureViewModel*** ist, obwohl keine ihrer Methoden verwendet wurden.

Um diese Kopplung aufzulösen, habe ich daher die Hilfsklasse ***TreeActionModelWrapper*** erstellt, welche die gleiche Aufgabe wie ***TreeModelWrapper*** übernimmt, ohne dabei die Bindung an die Structure View zu haben. Da sie nicht auf das B.E.M. Plugin zugeschnitten ist, befindet sie sich im Package ***com.holtkamp.intellij.common.tree***.

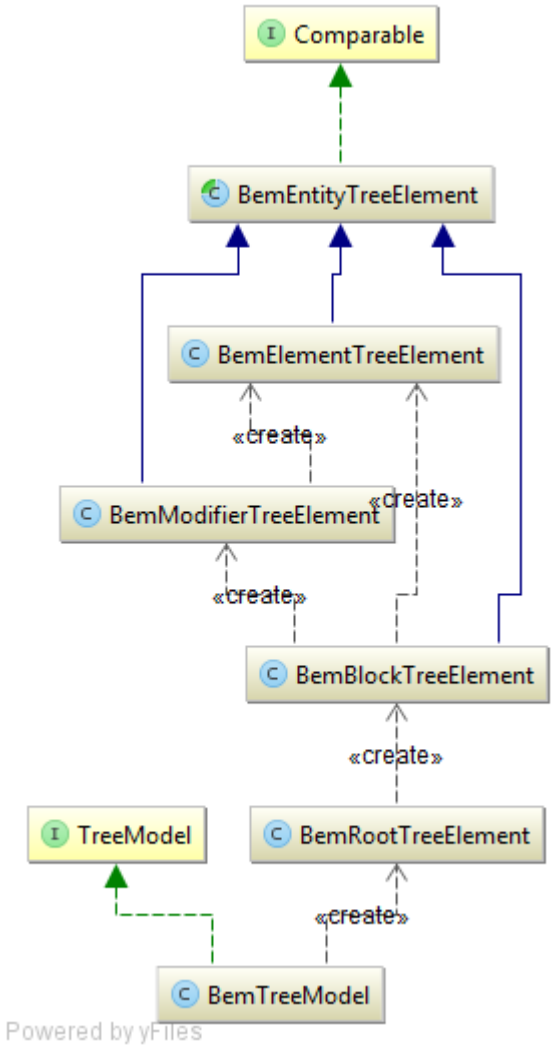


Abbildung 33: Klassendiagramm der Knoten des B.E.M. Tool-Windows

Das Model, welches von dem ***TreeActionModelWrapper*** umschlossen wird, liefert als Wurzel-Element eine Instanz von ***BemRootTreeElement*** zurück, welches abhängig von der aktiven Editor-Instanz einen ***BemAggregator*** erzeugt (vgl. Kapitel 4.2.6.1). Zur Darstellung der einzelnen Baum-Knoten ergibt sich die in Abbildung 33 zu sehende Klassenstruktur.

Für die Implementierung der einzelnen Funktionen in der Toolbar konnte ich mit Ausnahme der Sortierung auf vorhandene Implementierungen zurückgreifen und entsprechende Methoden am ***AbstractTreeBuilder*** und an der ***SmartTreeStructure*** Klasse aufrufen.

Einzig für die Sortierfunktion auf B.E.M Entitäten musste eine Klasse ***BemEntityTreeElementSorter*** erstellt werden, die in die Liste der Implementierungen des Interfaces ***Sort*** die das ***BemTreeModel*** zurückliefert, eingefügt wurde.

Die Klasse selbst liefert neben Informationen zur Darstellung des Sorters als Action einen ***Comparator***. Da in der Baumstruktur als Kind-Knoten nur Entitäten von ***BemEntityTreeElement*** auftauchen, kann man die Typprüfung zu Beginn der Implementierung auch weglassen, allerdings ziehe ich es vor, die Implementierung so generisch zu halten, dass man beliebige andere Einträge in den Baum einfügen könnte, ohne dass es hier zu Problemen kommt. Unabhängig davon entscheidet dann - sofern die Typsicherheit gewährleistet ist - die Implementierung von ***BemEntityTreeElement*** über die Reihenfolge. Hier gilt die grundsätzliche Reihenfolge: Block vor Element vor Modifier. Bei Gleichheit entscheidet ein String-Vergleich des entsprechenden Bezeichners.

Da die Implementierung der Aktualisierung wie in Kapitel 3.3.3.2 nicht zwingend automatisch erfolgen musste, habe ich zugunsten der Performance entschieden und einen entsprechenden Button in die Toolbar eingebaut, der eine manuelle Aktualisierung auslöst.

Anhang 6 zeigt schließlich das fertige Tool-Window.

4.2.7 Actions zur Generierung von B.E.M. Entitäten

Mit Hilfe von Actions lassen sich wie in Kapitel 2.4.5 beschrieben neue Einträge in Menüs und Toolbars vornehmen. Da das Menü erweitert werden soll, welches mit ***Alt*** + ***Einfg*** aufgerufen wird (vgl. Kapitel 3.3.4.1), ließ sich durch Betrachtung des Quellcodes von IDEA herausfinden, dass hierfür die ***GenerateGroup*** als ***group-id*** angegeben werden muss, wenn mit Hilfe des Tags ***add-to-group*** die Action einer Gruppe zugewiesen wird.

Jede der zu generierenden B.E.M. Entitäten stellt sich in einer Implementierung der Klasse ***AnAction*** dar. Da die Klasse selbst keine Methode hat, die den dargestellten Titel oder die Beschreibung setzen, musste ich während der ***update*** Methode die in ***AnActionEvent*** enthaltene ***Presentation*** Instanz entsprechend anpassen. Dies geschieht in einer gemeinsamen, abstrakten Basisklasse ***BemCreateEntityRulesetAction***, welche erwartet, dass alle ableitenden Klassen ***getTitle*** und ***getDescription*** implementieren.

Die Logik, die bestimmt, ob die Action überhaupt angezeigt werden soll, wird ebenfalls dort implementiert, da sie für alle Ausprägungen gleich sein soll. Sie prüft die Voraussetzung, dass die Action innerhalb einer Datei ausgelöst wurde und diese Datei in der CSS-Sprache geschrieben ist.

Im Folgenden wird die Logik bei der Generierung von B.E.M. Entitäten beschrieben, die die Vorbelegung der einzelnen Bezeichner festlegt. Darauf aufbauend wird dann die Idee hinter

dem Ansatz der generieren Live-Templates erläutert. Abschließend wird die Implementierung der ***actionPerformed*** Methode beschrieben.

4.2.7.1 Ermittlung der Vorbelegung von Bezeichnern

Die Kern-Logik für die Ermittlung der Vorbelegungen findet sich im Paket ***com.holtkamp.intellij.bemplugin.suggestions***.

Die beiden Interfaces ***BemIdentifierSuggestionStrategy*** und ***BemIdentifierSuggestion*** sind eng miteinander verknüpft: Während erstere als einzige Methode ***apply*** erwartet, stellt letztere den Rückgabewert der Methode in Abhängigkeit zu einem übergebenen ***BemAggregator*** (vgl. Kapitel 4.2.6.1) dar. Hierfür müssen die Methoden ***getBlockIdentifier***, ***getElementIdentifier*** und ***getModifierIdentifier*** implementiert werden.

Hierdurch wurden diverse Strategien implementiert, die Vorschläge für Belegungen von festgelegten Bezeichnern machen:

- **DefaultValueStrategy**
  Bestimmt die Vorschläge anhand von festen Werten, ignoriert also den übergebenen ***BemAggregator***.

- **BemAnyBlockStrategy**
  Initialisiert den Block Identifier mit dem Block-Bezeichner der ersten B.E.M. Entität, die in dem übergebenen ***BemAggregator*** gefunden wird. Der Rest wird mit festen Werten belegt.

- **BemAnyElementStrategy**
  Initialisiert den Block und den Element Identifier mit dem Block-Bezeichner und dem Element-Bezeichner aus dem ersten B.E.M. Element, das in dem übergebenen ***BemAggregator*** gefunden wird.

- **BemAnyModifierStrategy**
  Analog zu ***BemAnyElementStrategy***, nur für Modifier.

- **BemAnyModifiedElementStrategy**
  Analog zu ***BemAnyElementStrategy***, nur, dass das Element in einem Modifier sein muss. Der Modifier-Bezeichner wird ebenfalls übernommen.

Die hier definierten Strategien werden von den einzelnen ***AnAction***-Implementierungen verwendet, um dann die Vorbelegungen der Variablen in den Live-Templates zu bestimmen. Da sie aber keinerlei Bindung an die Actions haben, ließe sich eine Strategie auch für beliebige andere Anwendungsfälle verwenden.

4.2.7.2 Generierung von Live-Templates

Live-Templates können nicht nur über die von IDEA bereitgestellte Benutzungsoberfläche erzeugt werden sondern mit Hilfe der Plugin-API auch programmatisch.

Während Live-Templates nur den selektierten Text verwenden können, um Informationen für die Vorbelegung zu erhalten, ist ein Ansatz mit Actions, die Live-Templates generieren, viel flexibler:

- Der gesamte Kontext, in dem sich der Cursor befindet, kann berücksichtigt werden.
- Da nichts selektiert werden muss, wird auch nichts automatisch durch das Template ersetzt
- Der Cursor kann sogar noch nachjustiert werden, zum Beispiel, wenn das Live-Template innerhalb eines Kommentar-Blocks aufgerufen wurde.

4.2.7.3 Ausführung der Actions

An dieser Stelle soll die Implementierung der ***performAction*** Methode beschrieben werden. Wie zu Beginn des übergreifenden Kapitels beschrieben, gibt es eine abstrakte Klasse ***BemCreateEntityRulesetAction*** auf die sich alle konkreten Implementierungen stützen.

Während die ***performAction*** Methode hier implementiert wird, implementieren die erweiterten Klassen nur noch eine Menge von abstrakten Methoden, um die Strategie zur Findung von Vorbelegungen anzugeben und das Template-Segment, dass den CSS-Selektor abbildet, in das Live-Template einzufügen.

Generell gibt es bei der Ausführung der jeweiligen Aktion folgenden Ablauf:

- Prüfung, ob die Voraussetzungen für die Ausführung der Aktion noch gegeben sind
- Ermittlung des PSI-Elements, welches sich an der aktuellen Cursor-Position befindet
- Sofern ein PSI-Element gefunden wurde:
    - Erzeugung eines ***BemAggregator***s für einen CSS-Selektor oder eine CSS-Regel, wenn das PSI-Element selbst ein CSS-Selektor oder eine CSS-Regel ist oder sich in einem dieser befindet.
    - Anpassen der Cursor-Position, damit das Live-Template nicht innerhalb eines Kommentar-Blocks oder innerhalb einer CSS-Regel gestartet wird
- Ermitteln der Vorschläge für einzelne Bezeichner anhand der Strategien, die die konkrete Aktion mit Hilfe der Methode ***getStrategies*** übergibt. Wenn ein ***BemAggregator*** erzeugt wurde, wird dieser mit übergeben. Es gewinnt die Aktion, die zuerst einen Vorschlag macht.
- Scrollen an die aktuelle Cursor-Position

- Erzeugung des Live-Templates unter Einbettung des von der konkreten Action durch ***addBemSelektorSegment*** definierten Selektor-Segments.
- Starten des Live-Templates

Die Methoden ***getStrategies*** und ***addBemSelektorSegment*** werden von den einzelnen Actions folgendermaßen definiert:

Action-Klasse	**Strategien**	**Selektor-Segment**
CreateBlockRulesetAction	AnyBlockStrategy DefaultValueStrategy	.{block-ident}
CreateElementRulesetAction	AnyElementStrategy AnyBlockStrategy DefaultValueStrategy	.{block-ident}__{element-ident}
CreateModifierRulesetAction	AnyModifierStrategy AnyBlockStrategy DefaultValueStrategy	.{block-ident}--{modifier-ident}
CreateModifiedElementRulesetAction	AnyModifiedElementStrategy AnyModifierStrategy AnyElementStrategy AnyBlockStrategy DefaultValueStrategy	.{block-ident}--{modifier-ident} .{block-ident}__{element-ident}

Das Selektor-Segment wurde im Pseudo-Code dargestellt, dabei sind Variablen durch geschweifte Klammern abgebildet. Außerdem wurde der „Bem“-Prefix ausgelassen, um die Übersichtlichkeit zu erhalten.

Der in Anhang 7 befindliche Screenshot zeigt ein Live-Template, welches durch Ausführung der ***BemCreateModifiedElementRulesetAction*** aus dem Kontext der darüber liegenden Regel generiert wurde.

4.3 Test

Die Tests für das B.E.M. Plugin teilen sich in zwei Gruppen auf: manuelle Tests und automatische Tests. Funktionen, die die Benutzungsoberfläche betreffen, lassen sich zwar automatisch testen, allerdings muss hier immer der Aufwand- und Nutzen von automatischen Tests abgewogen werden:

Während manuelle Tests bei jeder Änderung einen Zeitaufwand verursachen, dafür aber deutlich schneller zu beschreiben sind, sorgt die Erstellung von automatischen Tests viel Aufwand, dafür entfällt der Aufwand bei jeder Durchführung. Wenn es wenige große Änderungen an dem Plugin gibt, die die Oberfläche betreffen, wird nach einer bestimmten Zeit der initiale, zeitliche Aufwand für einen automatischen Test von dem des manuellen Test übertroffen.

Die Summe aller Tests ergibt keine vollständige Testabdeckung des B.E.M. Plugins, vielmehr dienen sie in diesem Kontext als Beispiel, wie man die einzelnen Funktionalitäten testen könnte.

4.3.1 Manuelle Tests

Ich habe mich für die Beschreibung von Testplänen für die Durchführung von manuellen Tests für die Benutzungsoberfläche entschieden, da dies die erste Version des Plugins ist und mit Ausblick auf weitere Features die Wahrscheinlichkeit groß ist, dass es noch viele Änderungen an der Oberfläche geben wird.

Dies betrifft Features aus allen User-Stories. Ein genereller Aufbau aller Testpläne wird in (Spillner, Koomen und Pol 2002) beschrieben, die folgenden Kapitel zeigen die konkreten Ausprägungen für die jeweiligen User-Stories. Da die Menüstruktur von IDEA sich je nach Betriebssystem leicht unterscheidet, gelten die Testpläne nur für das Betriebssystem Windows 7 64-Bit. Testpläne für andere Betriebssysteme sollen in dieser Arbeit nicht weiter betrachtet werden.

4.3.1.1 **Testplan: Syntax-Highlighting und Color-Settings-Page**

Testplanung:

Da man mit Hilfe der Color-Settings-Page auch das Syntax-Highlighting mit testet, kann man die Schritte für diese beiden User-Stories in einem Testplan zusammenfassen. Der Test deckt also die Richtigkeit der den Kapiteln 4.2.4 und 4.2.5 beschriebenen Implementierung ab. Da die Schnittstelle vorsieht, Farbeinstellungen auf Tokenebene zu definieren (vgl. Kapitel 2.4.4.3), beschränkt sich der Umfang auf das Testen des Highlighting eben dieser und prüft keine Randfälle ab, die sich aus der Komposition von Tokens ergibt. Dies ist Aufgabe der IntelliJ Plugin-API und es wird die Annahme getroffen, dass hierfür entsprechende Tests vorgenommen wurden.

Testvorbereitung:

- Für den Test muss IntelliJ IDEA 15.0.6 und das B.E.M. Plugin auf dem System installiert sein (vgl. Anhang 1).
- Das Testprojekt (vgl. Anhang 3) auf muss an einen Ort mit Schreibrechten kopiert werden.
- IntelliJ muss gestartet und das Testprojekt geladen werden.
- Die Sprache von IntelliJ IDEA muss auf English gestellt sein.

Testspezifikation:

Nr.	Beschreibung	Erwartete Ergebnisse
A1	Menüeintrag ***File*** – ***Project Settings*** auswählen Im Dialog links ***Editor*** – ***Colors & Fonts*** öffnen	Es gibt einen Eintrag ***B.E.M.***
A2	Eintrag ***B.E.M.*** auswählen	Rechts wird ein Editor für B.E.M. angezeigt. Liste hat die Einträge: - B.E.M. Block Identifier - B.E.M. Element Identifier - B.E.M. Element Separator - B.E.M. Modifier Identifier - B.E.M. Modifier Separator - Non B.E.M. In der Beispielansicht sind B.E.M. Entitäten zu erkennen
A3	***B.E.M. Block Identifier*** in der Liste markieren	Beispielansicht hebt 3 Einträge mit dem Text „bem-block“ hervor Darstellung entspricht der neben der Liste ausgewählten Einstellungen
A4	***B.E.M. Element Identifier*** in der Liste markieren	Beispielansicht hebt den Text „element“ hervor. Darstellung entspricht der neben der Liste ausgewählten Einstellungen
A5	***B.E.M. Element Separator*** in der Liste markieren	Beispielansicht hebt den Text „__“ hervor. Darstellung entspricht der neben der Liste ausgewählten Einstellungen
A6	***B.E.M. Modifier Identifier*** in der Liste markieren	Beispielansicht hebt den Text „modifier“ hervor. Darstellung entspricht der neben der Liste ausgewählten Einstellungen
A7	***B.E.M. Modifier Separator*** in der Liste markieren	Beispielansicht hebt den Text „--" hervor. Darstellung entspricht der neben der Liste ausgewählten Einstellungen
A8	***Non B.E.M.*** in der Liste markieren	Beispielansicht hebt den Text „no__b--e--m" hervor. Darstellung entspricht der neben der Liste ausgewählten Einstellungen
A9	Alle Listeneinträge noch einmal in der Beispielansicht betrachten.	Alle Listeneinträge außer die beiden Separatoren sind in der Beispielansicht klar voneinander untereinander unterscheidbar.

Testdurchführung und -auswertung:
Testschritte A1-A9 waren für das installierte B.E.M. Plugin (vgl. Anhang 3) erfolgreich.

Testabschluss:
Ziele erreicht, Dialog ***Settings*** schließen und nächsten Testplan durchführen.

4.3.1.2 Testplan: Spracheinbindung und B.E.M. Tool-Window

Testplanung:

In diesem Testplan wird die Injektion der Sprache B.E.M. in eine CSS-Datei getestet (vgl. Kapitel 4.2.3). Außerdem soll die Funktionalität des B.E.M. Tool-Windows sichergestellt werden (vgl. Kapitel 4.2.6). Hierzu wird eine CSS-Datei bereitgestellt, welche das Beispiel aus Kapitel 3.3.3.3 aufgreift, an Hand dessen das Tool-Window beschrieben wurde.

Testvorbereitung:

Der Test wird im Anschluss an den vorherigen Testplan durchgeführt, somit wird erwartet, dass das Testprojekt in IntelliJ IDEA geöffnet ist.

Testspezifikation:

Nr.	Beschreibung	Erwartete Ergebnisse
B1	Wenn offene Editor-Fenster da sind, diese schließen	Es gibt in der unteren linken Ecke neben ***Favorites*** einen Eintrag ***B.E.M.***
B2	Eintrag ***B.E.M.*** neben ***Favorites*** anklicken	B.E.M. Tool-Window wird eingeblendet Titel zeigt BEM an Es gibt 4 Buttons in der Toolbar und 2 Separatoren und folgender Reihenfolge: - Refresh - Separator - Sort By Identifier - Separator - Expand All - Collapse All Die Buttons haben ein passendes Symbol und einen Tooltip Unter der Toolbar steht der Text ***Nothing to show***
B3	Datei ***B.css*** öffnen	Editorfenster geht auf In CSS-Selekoren werden die Texte ***box*** und ***example*** unterstrichen dargestellt.
B4	***Refresh*** anklicken	Unter der Toolbar ist eine Baumstruktur dessen erste Ebene zu sehen ist Beide Knoten sind aufklappbar
B5	***Expand All*** anklicken	Alle Knoten werden aufgeklappt Der gesamte Baum entspricht Abbildung 25 von Seite 64 hinsichtlich der Bezeichner und Symbole seiner Knoten (Aufklapper können sich unterscheiden)
B6	***Collapse All*** anklicken	Alle Knoten sind wieder zugeklappt
B7	***Sort By Identifier*** anklicken	Der Button bleibt gedrückt Der Baum lädt neu
B8	***Expand All*** anklicken	Die Knoten sind innerhalb ihrer Hierarchie nach Namen sortiert
B9	***Sort By Identifier*** anklicken	Der Button ist nicht mehr gedrückt

		Der Baum lädt neu
B10	***Expand All*** anklicken	Die Knoten sind nun nicht mehr innerhalb ihrer Hierarchie nach Namen sortiert

Testdurchführung und -auswertung:

Testschritte B1-B10 waren für das installierte B.E.M. Plugin (vgl. Anhang 3) erfolgreich.

Testabschluss:

Ziele erreicht, ***B.E.M. Toolwindow***, Editor für die Datei ***B.css*** schließen und nächsten Testplan durchführen.

4.3.1.3 Testplan: Actions zur Generierung von B.E.M. Entitäten

Testplanung:

Hier soll die Generierung von B.E.M. Entitäten über Actions getestet werden (vgl. Kapitel 4.2.7). Das Ziel soll nicht sein, die Funktionalität von Live-Templates in IDEA sicherzustellen, sondern es soll geprüft werden, ob die Actions eingeblendet werden und ein sinnvoller Wert für die Vorbelegung der Bezeichner vorgeschlagen wird.

Testvorbereitung:

Der Test wird im Anschluss an den vorherigen Testplan durchgeführt, somit wird erwartet, dass das Testprojekt in IntelliJ IDEA geöffnet ist.

Testspezifikation:

Nr.	Beschreibung	Erwartete Ergebnisse
C1	Datei ***C.css*** öffnen	Editor für die Datei öffnet sich Anfang von einigen CSS-Klassennamen wird unterstrichen dargestellt.
C2	Cursor in der letzten (leeren) Zeile der Datei platzieren Tastenkombination ***Alt*** + ***Einfügen*** drücken	Ein Popup-Menü öffnet sich Das Popup-Menü hat unter anderem folgende Einträge: - Create B.E.M. Block Ruleset - Create B.E.M. Element Ruleset - Create B.E.M. Modifier Ruleset - Create modified B.E.M. Element Ruleset
C3	***Create modified B.E.M. Element Ruleset*** auswählen	An der Position entsteht ein Live-Template Es ist mindestens eine Zeile Abstand zu dem vorherigen Ruleset Die Werte sind vorinitialisiert mit: - new-block - new-modifier - new-block

		- new-element new-block muss nur einmal geändert werden
C4	Live-Template abschließen	Der Cursor befindet sich in einer neuen Zeile innerhalb des Deklarationsblocks der neu erzeugten Regel
C5	4x Undo (***Strg*** + ***Z***) drücken	Generierter Code ist wieder entfernt
C6	Cursor in den Selektor von Regel 1 setzen Tastenkombination ***Alt*** + ***Einfügen*** drücken ***Create B.E.M. Element Ruleset*** auswählen	Der Cursor wird hinter Regel 1 gesetzt An der Position entsteht ein Live-Template Es ist mindestens eine Zeile Abstand zu dem vorherigen und nachfolgenden Ruleset Die Werte sind vorinitialisiert mit: - box - new-element
C7	***Strg*** + ***Z*** drücken	Live-Template ist wieder entfernt
C8	Cursor in den Deklarationsblock von Regel 2 setzen Tastenkombination ***Alt*** + ***Einfügen*** drücken ***Create B.E.M. Block Ruleset*** auswählen	Der Cursor wird hinter Regel 2 gesetzt An der Position entsteht ein Live-Template Es ist mindestens eine Zeile Abstand zu dem vorherigen und nachfolgenden Ruleset Der Wert ist vorinitialisiert mit: - box
C9	Live-Template abschließen	Der Cursor befindet sich in einer neuen Zeile innerhalb des Deklarationsblocks der neu erzeugten Regel
C10	Tastenkombination ***Alt*** + ***Einfügen*** drücken ***Create B.E.M. Modifier Ruleset*** auswählen	Der Cursor wird hinter Regel 2 gesetzt An der Position entsteht ein Live-Template Die Werte sind vorinitialisiert mit: - box - new-modifier
C11	***Strg*** + ***Z*** drücken	Live-Template ist wieder entfernt
C12	Cursor in den Deklarationsblock von Regel 2 setzen Tastenkombination ***Alt*** + ***Einfügen*** drücken ***Create modified B.E.M. Element Ruleset*** auswählen	Der Cursor wird hinter Regel 2 gesetzt An der Position entsteht ein Live-Template Es ist mindestens eine Zeile Abstand zu dem vorherigen Ruleset Die Werte sind vorinitialisiert mit: - box - highlighted - box - content

Testdurchführung- und -auswertung:

Testschritte C1-C12 waren für das installierte B.E.M. Plugin (vgl. Anhang 3) erfolgreich.

Testabschluss:

Ziele erreicht: Alle Funktionen, die manuelles Testen erfordern, wurden getestet. IDEA kann geschlossen und das Testprojekt gelöscht werden.

4.3.2 Automatische Tests

Das automatische Testen der Funktionalität des Plugins erfolgt aus IntelliJ IDEA heraus. Das Ziel der Tests ist es nicht, die Benutzungsoberfläche von IDEA zu testen (dies passiert im vorherigen Kapitel) sondern die Funktion der Java Klassen auf Model Ebene des B.E.M. Plugins sicherzustellen.

Als Framework zur Erstellung von Unit-Tests wird ***JUnit 4*** verwendet. Für die Implementierung von Integrations-Tests empfiehlt JetBrains die Nutzung einer sogenannten „headless environment“. Dies bedeutet, dass während der Tests keine Bildschirm- oder sonstige Grafikausgaben erfolgen. Ein solcher Ansatz wird in den meisten Fällen durch Netzwerkprotokolle gesteuert und überwacht.

Es wird zudem durch vorgefertigte Klassen seitens JetBrains ein Ansatz mit Fixtures vorgeschlagen. Dies bedeutet, dass eine Menge von Quelldateien nach der Ausführung eines Features eine Menge von Ausgabedateien erzeugt. Diese Ausgabedateien werden nun mit den erwarteten Ergebnissen verglichen. Ist das erwartete Ergebnis äquivalent zu dem erzeugten Ergebnis, war der Test erfolgreich.

Dieser Ansatz wird für die Verifizierung des aus der Parser Definition erzeugten PSI-Baums (vgl. Kapitel 3.4.1.1, 4.2.2) verwendet. Zur Generierung der Fixtures war die bereits in Kapitel 3.4.1.2 verwendete ***View PSI Structure*** Funktion sehr nützlich, da der dort ausgelesene PSI-Baum als Textdarstellung gespeichert werden kann. Diese Text-Darstellung kann bei Nutzung der Klasse ***ParsingTestCase*** als Basis-Testklasse verwendet werden, um die Fixture mit dem tatsächlichen Ergebnis zu vergleichen.

4.3.2.1 Einrichtung der Testumgebung

Zum Ablegen der Test Klassen wurde ein Verzeichnis ***test*** angelegt und in IDEA als ***Test Source Folder*** markiert. Zudem befinden sich die Fixtures in einem Verzeichnis ***testData***, welches als ***Test Resource Folder*** markiert wurde.

4.3.2.2 Ausführung der Tests

Die Tests können bei Auswahl des B.E.M. Projekts mit dem Menüeintrag ***Run*** – ***Run ‘all in bem-plugin’*** ausgeführt werden.

5 Bewertung

Die folgenden Kapitel bewerten das Ergebnis der letzten Kapitel hinsichtlich des verwendeten Entwicklungsmodells, der Praxistauglichkeit des erstellen Plugins und dessen mögliche Übertragung auf andere Entwicklungsumgebungen.

5.1 Entwicklungsmodell

Bei der Entwicklung des B.E.M. Plugins kam mir die enorm umfangreiche und mächtige Plugin API von IntelliJ IDEA sehr entgegen. Das große Problem, das sich bei der API eröffnet, ist die nur in einigen Teilen vorhandene Code-Dokumentation und die auf noch weniger Teile beschränkte Architektur-Dokumentation:

- Für Features, die nicht in einem Tutorial beschrieben waren, musste ich sehr häufig andere Open-Source Implementierung zu Rate ziehen, um mir dort anzusehen, wie ein ähnliches Problem dort gelöst wurde.
- Außerdem musste ich mit Hilfe des integrierten Decompilers den nicht durch die Community-Edition bereitgestellten Quellcode von IntelliJ IDEA betrachten (z.B. für das CSS-Plugin), um das Verhalten zu analysieren und daraus abzuleiten, wofür bestimmte Interfaces, Klassen oder ganze Schnittstellen da sind und wie sie funktionieren.
- Dies kostet nicht nur viel Zeit, sondern schafft eine gewisse Unsicherheit, ob man für alle Ansätze die richtige Lösung gewählt hat, da zum einen das Vorgehen nicht beschrieben war und zum anderen die Alternativen nicht ersichtlich wurden.

Als sehr hilfreich hingegen hat sich das Schreiben von User-Stories herausgestellt. Hierdurch war ich gezwungen, einen oder mehrere andere Blickwinkel auf das Vorhaben einzunehmen und konnte noch besser über die Anforderungen an bestimmte Features nachdenken. Hierdurch entstand auch die Idee, Live-Templates für das Generieren von Actions zu verwenden.

5.2 Praxistauglichkeit

Da in dieser Ausarbeitung kein Praxistest für das Plugin vorgesehen war, besteht zudem keine Sicherheit, ob das Plugin realen Menschen bei der täglichen Entwicklung wirklich helfen kann.

5.2.1 Usability-Tests

Ein Praxistest kann zum Beispiel in Form eines Usability-Tests durchgeführt werden. Für diesen Fall würden hierbei dann Testpersonen benötigt, die mit IntelliJ IDEA vertraut sind und Erfahrungen in der Frontend-Entwicklung mit HTML und CSS haben.
Da das Plugin bei der Anwendung des Patterns unterstützen soll, ist eine Kenntnis des Patterns nicht zwingend erforderlich. Wie auch in Kapitel 3.3 könnte es verschiedene Test-Gruppen gehen, die sich aus der Vorkenntnis über das Pattern zusammensetzen. Allen Testpersonen könnten vor Beginn des Tests eine zusätzliche Einweisung erhalten, in der erklärt wird, wie das B.E.M. Pattern anzuwenden ist. Da es wie in Kapitel 2.3.1 bereits erwähnt, verschiedene syntaktische Varianten des B.E.M. Patterns gibt, ist dies auf jeden Fall erforderlich, damit alle Testpersonen das gleiche Verständnis aufbringen.

Die Durchführung des Tests würde mit Hilfe eines Testplans erfolgen, der die einzelnen Funktionen des Plugins auf Nutzbarkeit untersucht. So kann der Testperson zum Beispiel gesagt werden, dass sie die mit Hilfe des B.E.M. Tool-Windows herausfinden soll, ob der Modifier eines bestimmten Blocks Einfluss auf die Darstellung eines bestimmten Elements hat.

In der Regel werden sowohl die Konversation als auch der sichtbare Bildschirmbereich des Users in einem Usability-Tests aufgezeichnet. Als Unterstützung bietet sich hier auch ein Eye-Tracker an, der weitere Einblicke darüber liefern könnte, wo der User bestimmte Funktionen erwartet bzw. nach ihnen sucht.

Abgeschlossen wird ein solcher Test mit dem Ausfüllen eines Fragebogens, mit denen der Tester unabhängig von den Beobachtungen des Testers seinen Eindruck über das Plugin schildern kann.

5.2.2 Sammeln von statistischen Daten

Eine weitere Möglichkeit wäre die Sammlung und Auswertung von statistischen Daten. So sammelt IntelliJ IDEA zum Beispiel – sofern vom jeweiligen Benutzer erlaubt – Benutzungsstatistiken. Falls es eine Möglichkeit gibt, dies für eigene Plugins zu verwenden,

würde hier viel Aufschluss darüber gewonnen werden können, wie das Plugin verwendet wird.

Mögliche Informationen wären zum Beispiel:

- **Anzahl an Installationen**
  Dies erschafft Klarheit darüber, wie bekannt das Plugin ist.

- **Anzahl der Deinstallationen (mit Dauer, wie lange das Plugin genutzt wurde)**
  Dies gibt einen sehr groben Einblick darüber, wie zufrieden die Benutzer mit dem Plugin sind (natürlich wird die Statistik dadurch verfälscht, dass nicht alle Deinstallationen aus Unzufriedenheit erfolgen)

- **Klickpfade**
  Durch Analyse der Klickpfade bei der Nutzung des Plugins können immer wieder erfolgte Muster erkannt werden, die gegebenenfalls zu einer Optimierung führen können.

- **Fehlerberichte**
  Durch die Möglichkeit, Fehlerberichte einzusenden, können Bugs in dem Plugin erkannt und behoben werden.

5.3 Übertragung auf andere Entwicklungsumgebungen

Der Übertragung auf andere Entwicklungsumgebungen steht im Grunde nichts entgegen. Der Wunsch nach Unterstützung durch die IDE, der mit den User-Stories veranschaulicht wurde, bleibt gleich, egal, ob man mit IntelliJ IDEA, Netbeans, Eclipse oder einer völlig anderen Umgebung arbeitet.

Bedingt durch den großen Funktionsumfang der IntelliJ API, die auch einen hohen Vorfertigungsgrad anbietet, sind die einzelnen Implementierungen allerdings auch sehr eng an IntelliJ gekoppelt. Abgesehen von einigen Modellklassen, müsste jedes Feature entsprechend der API der jeweiligen IDE neu implementiert werden. Dies setzt natürlich voraus, dass ein ähnlicher Ansatz wie Sprachinjektion überhaupt möglich ist, sonst würde es sogar nötig werden, die gesamte CSS Datei selbst zu parsen.

6 Zusammenfassung und Ausblick

Das Ergebnis dieser Bachelorarbeit ist ein lauffähiges und getestetes B.E.M. Plugin für die Entwicklungsumgebung IntelliJ IDEA.

Es stehen folgende Funktionen zur Verfügung:

- Einführung des B.E.M. Patterns als eigene Sprache
- Integration der Sprache B.E.M. in die von IntelliJ IDEA bereitgestellte Sprache CSS
- Syntax-Highlighting der einzelnen Entitäten des B.E.M. Patterns
- Anpassung des Syntax-Highlighting nach den Präferenzen des Benutzers
- Eine übersichtliche Darstellung der B.E.M. Struktur einer Datei in einem Tool-Window
- Generierung von CSS Klassen, die einzelne B.E.M. Entitäten repräsentieren

Neben diesem funktionalen Ergebnis gibt es auch ein persönliches Ergebnis für mich:
Ich habe zum ersten Mal ein Plugin für IntelliJ IDEA geschrieben. Das neu erlangte Wissen gibt mir viele Ideen für weitere Funktionen, die man innerhalb der Entwicklungsumgebung bereitstellen kann - nicht nur für die Integration des B.E.M. Patterns.

Allerdings ist das neu erlangte Wissen nicht nur auf diesen Bereich beschränkt: Durch die Analyse der Plugin API habe ich viele Einblicke in die Architektur von IDEA bekommen, die ich auch auf völlig andere Bereiche übertragen kann.

Die in dieser Ausarbeitung implementierten Funktionalitäten stellen eine Grundlage dar, die sich nach Belieben ausbauen lässt. So wären noch weitere Funktionen denkbar, die die Arbeit mit dem Pattern weiter unterstützen können:

- **Automatische Anordnung von CSS Regeln**
  Alle Regeln, die sich auf denselben Block beziehen, könnten unter Einbehaltung ihrer ursprünglichen relativen Position untereinander angeordnet werden.
  Dies birgt ein Risiko, da man sich sicher sein muss, dass B.E.M. korrekt eingehalten wurde und nicht durch falsche oder nicht projektweite Nutzung indirekte Abhängigkeiten von Blocks durch die Reihenfolge der Styling-Deklarationen entstanden sind, welche bei einer Neuanordnung verändert wird.

- **Integration in die Sprache HTML**
  Die Einbindung des B.E.M. Plugins in HTML wurde bisher nicht vorgenommen. Denkbar wären zum Beispiel neben Syntax-Highlighting auch Code-Completion für B.E.M. Element Klassennamen, wenn das ***class*** Attribut eines HTML Element unterhalb eines Blocks verändert werden soll, oder analog dazu die Auswahl von B.E.M. Modifier Klassennamen am gleichen HTML Element an der der Block festgelegt wurde.

- **Usages Provider**
  Es existiert zwar ein Usages Provider für CSS Klassennamen, allerdings könnte dieser erweitert werden, wenn man einen B.E.M. Block Klassennamen auswählt. So könnte es eine weitere Auflistung für B.E.M. geben, in der alle HTML Elemente angezeigt werden, die den Block, eines seiner Elemente oder einen Modifier verwenden.

- **Validierung der Verwendung in HTML**
  In Kapitel 2.3.1 wird beschrieben, in welchem Verhältnis B.E.M. Klassennamen im DOM erlaubt sind.
  Das Plugin könnte entsprechende Validity-Handler in IDEA einbinden, um den Benutzer auf eine falsche Anwendung von B.E.M. hinzuweisen.

- **Validierung von CSS Selektoren**
  Aufbauend auf der vorherigen Idee, könnten dann auch meine aus Kapitel 3.1.2 vorgeschlagenen „Stricter B.E.M.“ Erweiterungen als optionale Validierung in das Plugin einfließen.

Ich werde die Arbeit an dem B.E.M. Plugin auf jeden Fall als persönliches Projekt fortführen. Einerseits, um noch mehr Einblicke in die Plugin API zu bekommen, andererseits, weil ich selbst Anwender des B.E.M. Patterns bin und mich das Plugin schon jetzt in meinen Frontend Projekten stark unterstützt.

A Literaturverzeichnis

Appel, Andrew W., und Jens Palsberg. *Modern Compiler Implementation in Java.* United Kingdom: Press Syndicate of the University of Cambridge, 2012.

Bos, Bert, Tantek Çelik, Ian Hickson, and Håkon Wium Lie. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification.* April 12, 2016. https://www.w3.org/TR/2011/REC-CSS2-20110607/ (accessed April 24, 2016).

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, und Michael Stal. *Pattern-Oriented Software Architecture.* 1996.

Çelik, Tantek, Elika J. Etemad, Daniel Glazman, Ian Hickson, Peter Linss, und John Williams. *Selectors Level 3.* 29. September 2011. https://www.w3.org/TR/2011/REC-css3-selectors-20110929/ (Zugriff am 10. Mai 2015).

Fischer, Peter, und Peter Hofer. *Lexikon der Informatik.* Springer, 2008.

Friedman, Vitaly, et al. *The Smashing Book #4.* Freiburg, Germany: Smashing Magazine GmbH, 2013.

Gamma, Erich, Richard Helm, Ralph Johnson, und John Vlissides. *Design Patterns - Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software.* 2015.

Hickson, Ian, et al. *HTML5.* 28. Oktober 2014. https://www.w3.org/TR/2014/REC-html5-20141028/ (Zugriff am 10. Mai 2015).

ISO/IEC 14977. 12. Dezember 1996.

JetBrains Development Team. *Quellcode der IntelliJ IDEA Community Edition 15.0.6.* 2016.

JetBrains Documentation Team. *IntelliJ Platform SDK.* 18. Februar 2016. http://www.jetbrains.org/intellij/sdk/docs/index.html (Zugriff am 12. Mai 2016).

JetBrains Homepage Team. *Technology-leading software development firm specializing in the creation of intelligent development tools.* 2016. https://www.jetbrains.com/company/ (Zugriff am 14. Mai 2016).

Klein, Gerwin, Steve Rowe, und Régis Décamps. *JFlex - User´s Manual.* 20. April 2015. http://jflex.de/manual.html (Zugriff am 12. Mai 2016).

Lahres, Bernhard, und Gregor Rayman . *Objektorientiere Programmierung.* 2009. http://openbook.rheinwerk-verlag.de/oop/ (Zugriff am 12. Dezember 2015).

Lemay, Laura, und Rogers Cadenhead. *Java in 21 Tagen.* München: Markt & Technik, Buch- und Software-Verlag, 2000.

Lie, Wium Håkon, und Bert Bos. *Cascading Style Sheets, level 1.* 17. Dezember 1996. https://www.w3.org/TR/2008/REC-CSS1-20080411/ (Zugriff am 10. Mai 2015).

Nayak, Supratim. *HYDRATTZ (Supratim Nayak) - DeviantArt.* 2008. http://hydrattz.deviantart.com/ (Zugriff am 12. Mai 2016).

Smith, Michael. *HTML: The Markup Language (an HTML language reference).* 28. Mai 2013. https://www.w3.org/TR/2013/NOTE-html-markup-20130528/ (Zugriff am 12. Mai 2016).

Spillner, Andreas, Tim Koomen, und Martin Pol. *Management und Optimierung des Testprozesses.* Berlin: Dpunkt.Verlag GmbH, 2002.

van Kesteren, Anne, Aryeh Gregor, Ms2ger, Alex Russell, und Robin Berjon. *W3C DOM4.* 19. November 2015. https://www.w3.org/TR/2015/REC-dom-20151119/ (Zugriff am 10. Mai 2016).

B Anhang

1) Starten Sie die Installation von IntelliJ, die sich auf dem beiliegendem Datenträger im Root-Verzeichnis befindet und den Namen ***ideaIU-15.0.6.exe*** hat.
2) Folgenden Sie den Installationsschritten und schließen Sie die Installation ab.
3) Für die Ultimate-Edition von IDEA wird eine entsprechende Seriennummer benötigt. Dieser lässt sich teilnehmende Hochschulen hier beantragen: https://www.jetbrains.com/shop/eform/students
4) Starten Sie IDEA.
5) Da noch kein Projekt angelegt wurde, erscheint der ***Welcome to IntelliJ IDEA***-Dialog. Wählen Sie den Menüeintrag ***Configure*** – ***Plugins*** aus.
6) Im darauf folgenden ***Plugins***-Dialog verwenden Sie ***Install Plugin From Disc***.
7) Wählen Sie die Datei ***bem-plugin.jar*** aus, welche sich auf dem beiliegenden Datenträger befindet.
8) Schließen Sie die Installation des B.E.M. Plugins durch Verwendung von ***Restart IntelliJ IDEA*** ab.

Anhang 1: Installations-Anleitung für IntelliJ IDEA und das B.E.M. Plugin unter Windows

```
package com.holtkamp.intellij.bemplugin.lang;

import com.intellij.lexer.FlexLexer;
import com.intellij.psi.tree.IElementType;
import com.holtkamp.intellij.bemplugin.psi.BemTypes;
import com.intellij.psi.TokenType;

%%

%class BemLexer
%implements FlexLexer
%unicode
%function advance
%type IElementType
%eof{  return;
%eof}

// used macros from css language syntax
// @see https://www.w3.org/TR/CSS2/syndata.html#tokenization
//
// because %%ignorecase cannot be used for macros, modified all asci character sets to also
allow the uppercase version
ident=[-]?{nmstart}{nmchar}*
nmstart=[_a-zA-Z]|{nonascii}|{escape}
nonascii=[^\0-\237]
escape={unicode}|\\[^\n\r\f0-9a-fA-F]
unicode=\\[0-9a-fA-F]{1,6}(\r\n|[ \n\r\t\f])?
nmchar=[_a-zA-Z0-9-]|{nonascii}|{escape}
nl=\n|\r\n|\r|\f
w=[ \t\r\n\f]*

// macros for B.E.M.
block_ident_first_chars=[-]{nmstart}
block_ident_first_char={nmstart}
other_ident_first_char={nmchar}

block_ident={ident}
other_ident={nmchar}+

element_separator=[_][_]
modifier_separator=[\-][\-]
any_separator={element_separator}|{modifier_separator}

whitespace=({nl}|{w})+

// states
%state COLLECT_BLOCK_IDENTIFIER
%state ELEMENT
%state COLLECT_ELEMENT_IDENTIFIER
%state MODIFIER
%state COLLECT_MODIFIER_IDENTIFIER

%%

{element_separator}                                                    { yybegin(ELEMENT);
return BemTypes.ELEMENT SEPARATOR; }
{modifier_separator}                                                   { yybegin(MODIFIER);
return BemTypes.MODIFIER_SEPARATOR; }
{block_ident}?{any_separator}{other_ident}?{any_separator}{other_ident}? { return
BemTypes.NO BEM; }
{block_ident}?{element_separator}                                      { yypushback(2);
yybegin(ELEMENT); return BemTypes.BLOCK IDENTIFIER; }
{block_ident}?{modifier_separator}                                     { yypushback(2);
yybegin(MODIFIER); return BemTypes.BLOCK_IDENTIFIER; }

<YYINITIAL> {block_ident_first_char}                                   { yypushback(1);
yybegin(COLLECT_BLOCK_IDENTIFIER); }
<ELEMENT> {other_ident_first_char}                                     { yypushback(1);
yybegin(COLLECT_ELEMENT_IDENTIFIER); }
<MODIFIER> {other_ident_first_char}                                    { yypushback(1);
yybegin(COLLECT_MODIFIER_IDENTIFIER); }

<COLLECT_BLOCK_IDENTIFIER> {block_ident}                               {
yybegin(YYINITIAL); return BemTypes.BLOCK IDENTIFIER; }
<COLLECT_ELEMENT_IDENTIFIER> {other_ident}                             {
yybegin(YYINITIAL); return BemTypes.ELEMENT_IDENTIFIER; }
<COLLECT_MODIFIER_IDENTIFIER> {other_ident}                            {
yybegin(YYINITIAL); return BemTypes.MODIFIER_IDENTIFIER; }

{whitespace}                                                           {
yybegin(YYINITIAL); return TokenType.WHITE SPACE; }
.                                                                      { return
TokenType.BAD_CHARACTER; }
```

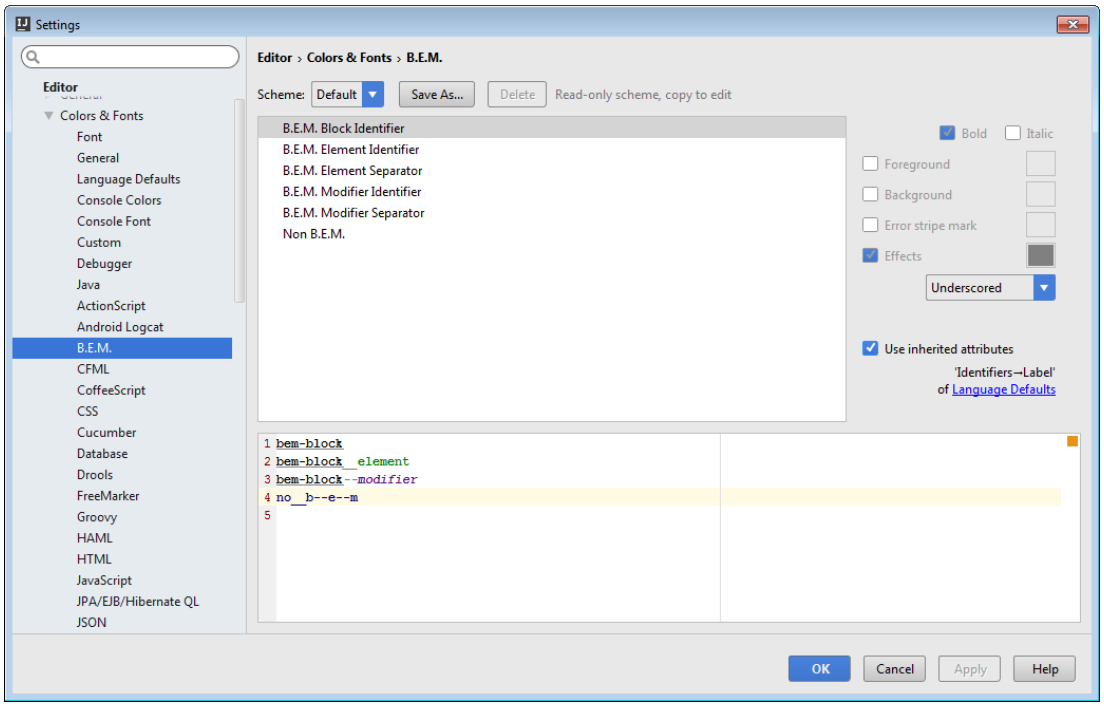
Anhang 2: Lexer für B.E.M. in JFlex Notation

- **/bem-plugin**
  Der Quellcode des B.E.M. Plugins
- **/intellij-community**
  Der Quellcode der IntelliJ IDEA Community Edition 15.0.6
- **/Quellen**
  Alle elektronischen Quellen
- **/test-project**
  Das Test-Projekt, welches für manuelle Tests verwendet wird
- **/Bachelorarbeit.pdf**
  Die Bachelorarbeit im PDF-Format
- **/bem-plugin.jar**
  Das B.E.M. Plugin in installierbarer Form
- **/ideaIU-15.0.6.exe**
  Installationsroutine des IntelliJ IDEA 15.0.6 Ultimate Edition für Windows
- **/jdk-8u74-windows-x64.exe**
  Installationsroutine des JDK8 in der Windows 64-Bit Version

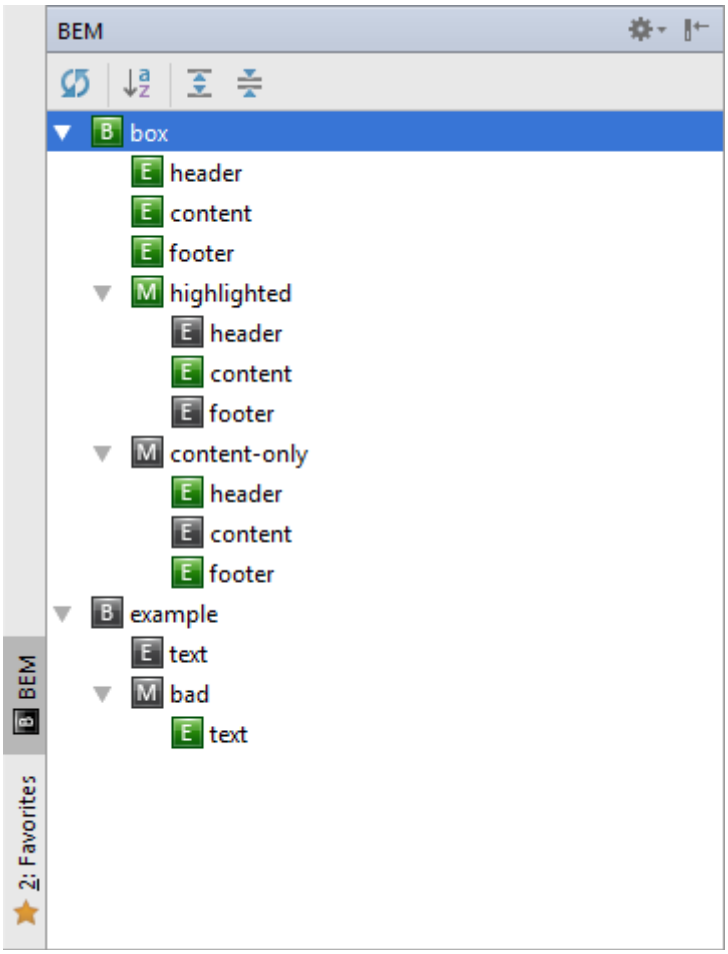
Anhang 3: Inhaltverzeichnis des beliegenden Datenträgers

```
.other__naming__pattern {
  display: none;
}

.grid {
  display: block;
  background-color: #919191;
}

.grid--2-columns {
  background-color: #BF8080;
}

.grid__row {
  display: block;
  font-size: 0;
}

.grid__col1,
.grid__col2 {
  display: inline-block;
  font-size: 1rem;
}

.grid__col1 {
  width: 100%;
}

.grid--2-columns .grid__col1 {
  width: 50%;
}

.grid--2-columns .grid__col2 {
  width: 100%;
}
```

Anhang 4: Screenshot des B.E.M. Syntax-Highlightings



Anhang 5: Screenshot der Color-Settings Page



Anhang 6: Screenshot des B.E.M. Tool-Window

```
.box--highlighted {
    border-color: red;
}

.box--highlighted .box__new-element {

}
```

Anhang 7: Screenshot vom Live-Template der BemCreateModifiedElementRulesetAction

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____