



Fachbereich
Elektrotechnik und Informatik



FACH
HOCHSCHULE
LÜBECK

University of Applied Sciences

Bachelorarbeit

Webanwendung zur Unterstützung des Lernens der Methode Semantisches Tableau

vorgelegt von: Tram Nguyen
Erstbetreuer: Prof. Dr. rer. nat. Andreas Schäfer
Zweitbetreuer: Prof. Dr.-Ing. Stefan Krause



Erklärung zur Bachelorarbeit

Ich versichere, dass ich die Arbeit selbständig, ohne fremde Hilfe verfasst habe.

Bei der Abfassung der Arbeit sind nur die angegebenen Quellen benutzt worden. Wörtlich oder dem Sinne nach entnommene Stellen sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, insbesondere dass die Arbeit Dritten zur Einsichtnahme vorgelegt oder Kopien der Arbeit zur Weitergabe an Dritte angefertigt werden.

Lübeck, den 30. August 2018

.....

(Unterschrift)

Inhalt

1	Einleitung	1
2	Analyse	2
2.1	Problemanalyse: Lastenheft	2
2.2	Anforderungsanalyse	4
2.2.1	Anwendungsfalldiagramm	4
2.2.2	Anwendungsfallbeschreibung	4
3	Grundlagen der Aussagenlogik	7
3.1	Syntax der Aussagenlogik	7
3.1.1	Alphabet der Aussagenlogik	7
3.1.2	Syntax aussagenlogischer Formeln	8
3.1.3	Bindungskonventionen	9
3.1.4	Baum-Notation von Formeln	9
3.1.5	Operator-Notation	10
3.1.6	Eine formale Grammatik für Formeln	11
3.2	Erfüllbarkeit und Tautologien	11
3.3	Semantisches Tableau	12
4	Frameworks	16
4.1	Antlr4	16
4.1.1	Hauptteilen von ANTLR	17
4.1.2	ANTLR JavaScript target	18
4.2	Bootstrap	19
4.2.1	Eigenständige mobile Website (Separate mobile Website)	19
4.2.2	Sich anpassendes Design (Responsive-Design)	20
4.2.3	Mobile-first	20
4.2.4	Bootstrap	21
4.3	Google chart tools	25
4.3.1	Organigramm	25
4.3.2	DataTable	27
4.3.3	Zeichnen des Diagramms	27
4.4	Jasmine	28
4.4.1	Wie benutzt man Jasmin?	28
4.4.2	Wie man mit Jasmine JS Tests prüft?	29
5	Architektur	31
5.1	Architekturdiagramm	31
5.2	Klassendiagramm	32

6	Implementierung	33
6.1	Struktur Dateisystem	35
6.2	FormulaGrammar.g4	37
6.3	Model	38
6.3.1	Formula	38
6.3.2	FormulaListener	41
6.3.3	TableauNode	43
6.3.4	Tableau	45
6.3.5	TableauForPropositionalLogic	46
6.4	View und Controller	47
6.4.1	View	47
6.4.2	Controller	55
7	Softwaretest	58
7.1	Konzept zum Softwaretest	58
7.2	Evaluation der Testfälle	59
7.3	Code Coverage	61
8	Zusammenfassung	63

1

Einleitung

Aussagenlogik ist immer ein grundlegendes Thema in der Informatik. Dabei ist die Ermittlung der Allgemeingültigkeit oder Erfüllbarkeit einer aussagenlogischen Formel unverzichtbar. Es gibt viele Möglichkeiten, dies zu tun, wobei die Verwendung der Wahrheitstafel recht aufwändig sein kann. Die Äquivalenzumformung liefert auch kein Verfahren, das sagt, welche Regel anzuwenden ist [Sch17]. Auch diese Methode ist schwer zu programmieren. Eine andere Methode ist einfacher und intuitiver: Eine Formel der Aussagenlogik kann als ein semantisches Tableau nach syntaktischen Regeln konstruiert werden. Aus dem derart konstruierten Tableau kann dann ermittelt werden, ob eine Formel erfüllbar oder allgemeingültig ist. Diese Methode kann auch einfach programmiert werden. Dies sind jedoch neue Inhalte, die seit dem WS 2017-2018 in den Lehrplan der Informatik 1 aufgenommen wurden. Um den Studierenden den Erwerb dieser Methode zu erleichtern, wird daher die Idee einer Webanwendung gebildet.

Das Hauptziel dieser Arbeit ist es, eine Webanwendung zu entwickeln. Es ermöglicht dem Benutzer, eine aussagenlogische Formel einzugeben und die Allgemeingültigkeit oder Erfüllbarkeit der Formel zu überprüfen. Zuerst prüft die Anwendung, ob die Formel syntaktisch korrekt ist. Wenn dies nicht der Fall ist, wird die Anwendung den Benutzer informieren, um es zu beheben. Die Anwendung zeigt dann die Konstruktionen für die Eingabeformel und für das Tableau an. Schließlich wird das Prüfergebnis auf dem Bildschirm angezeigt.

Der Inhalt dieser Arbeit gliedert sich in 6 Hauptteile. Die erste besteht darin, das Problem und die Anforderungen der Anwendung zu analysieren (Kapitel 2). Als nächstes lernt man etwas über die Grundlagen der Aussagenlogik (Kapitel 3) und man wählt die geeigneten Frameworks (Kapitel 4). Als nächstes soll die Architektur der Anwendung (Kapitel 5) erstellt und umgesetzt werden (Kapitel 6). Abschließend: Anwendungstests und Test-Bewertung (Kapitel 7)

2

Analyse

Dieses Kapitel handelt von der Problemanalyse sowie den Anforderungsdefinitionen der Anwendung.

2.1 Problemanalyse: Lastenheft

1. **Zielbestimmung** Es gibt ein Verfahren, das die Erfüllbarkeit einer aussagenlogischen Formel überprüft, die Tableau-Verfahren genannt wird. Dazu werden systematische Regeln angewendet und Formeln umgeschrieben. Dieses Verfahren kann auch als Programm implementiert und genutzt werden. Daher soll, um das Lehren und Lernen der Informatikmodule besser zu unterstützen, eine Software entwickelt werden, die vom Nutzer eine aussagenlogische Formel annimmt und prüft, ob diese Formel erfüllbar oder allgemeingültig ist. Das Ergebnis soll mittels eines semantischen Tableaus erstellt und angezeigt werden. Außerdem soll die Anwendung eine “Schritt für Schritt Lösung”- Funktion, wonach Studierende Schritt für Schritt den Aufbau eines Tableaus folgen können, bieten. Die Software soll als Web-Applikation eingeführt werden.

2. Produkteinsatz

/LE10/ Verwendung im Bereich Lehren und Lernen der Informatikmodule.

/LE20/ Die Nutzer sollen die Funktionalität über eine Webanwendung nutzen können.

/LE30/ Die Software soll ohne Server und Datenbank funktionieren können.

/LE40/ Zielgruppe der Software sind Studierende und die Dozierende.

3. Produktfunktionen

/LF10/ Eingabeformel überprüfen und anzeigen

- Die Software prüft, ob die Eingabeformel ein wohlgeformter Ausdruck der Aussagenlogik ist, sonst Wiederholung der Eingabe.

- Die Eingabeformel-Darstellung wird als Baumstruktur erstellt und in dem GUI angezeigt .

/LF20/ Allgemeingültigkeit überprüfen und Tableau darstellen

- Die Software erstellt die Negation von der Eingabeformel.
- Die Software mittels Tableau-Verfahren prüft, ob diese Negation erfüllbar ist (/LF30/).
- Wenn diese Negation nicht erfüllbar ist, ist die Eingabeformel allgemeingültig.
- Wenn diese Negation erfüllbar ist, ist die Eingabeformel nicht allgemeingültig.
- Die Tableau-Darstellung wird als Baumstruktur erstellt und in dem GUI angezeigt.

/LF30/ Erfüllbarkeit überprüfen und Tableau darstellen

- Die Software mittels Tableau-Verfahren prüft, ob Eingabeformel erfüllbar ist
- Die Tableau-Darstellung wird als Baumstruktur erstellt und in dem GUI angezeigt.

/LF40/ Schritt für Schritt Lösung anzeigen

- Ein Pop-up Fenster von der Schritt für Schritt Lösung wird geöffnet.
- Durch Anklicken eines Buttons kann das Tableau Stück für Stück dargestellt werden und ein Lösungshinweis für jeden Schritt angezeigt werden.

4. Produktdaten

- keine Produktdaten

5. Produktleistungen

/LL10/ Die Funktionen /LF20/ und /LF30/ darf nicht länger als 5 Sekunden Reaktionszeit benötigen

/LL20/ Bei fehlerhaften Eingaben erhält der Nutzer eine Fehlermeldung

/LL30/ Bei fehlerhaften Eingaben muss der Nutzer die Möglichkeit haben, eine Korrektur der Eingaben vorzunehmen.

6. Qualitätsanforderungen

/LQ10/ Vollständigkeit: Alle implementierten Funktionen werden benutzt. Alle referenzierten Funktionen werden implementiert.

7. Ergänzungen

- Die Software soll zunächst unter Chrome laufen, langfristig aber auch unter Firefox und weiteren Browsern.

2.2 Anforderungsanalyse

2.2.1 Anwendungsfalldiagramm

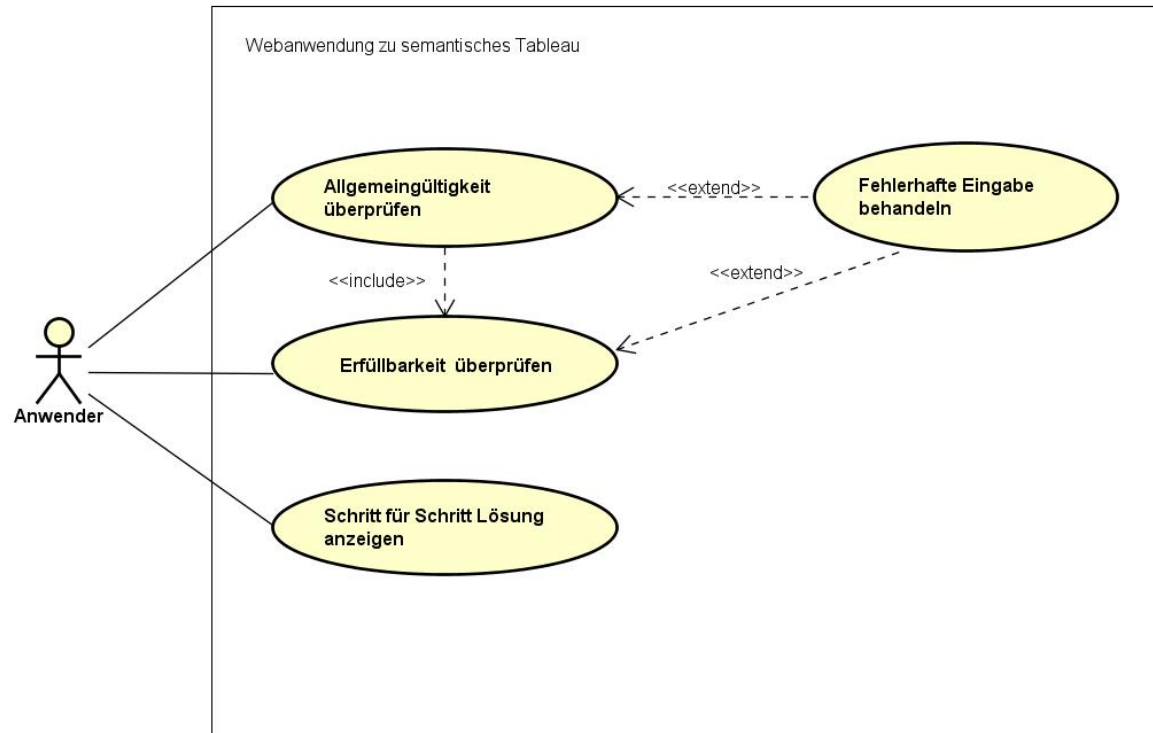


Abb. 2.1. Anwendungsfalldiagramm

2.2.2 Anwendungsfallbeschreibung

- **Anwendungsfall:** *“Allgemeingültigkeit überprüfen”*

Akteure:

Anwender (initiiert den Anwendungsfall)

Auslöser:

Anwender drückt auf “Tautologie”

Anfangsbedingungen:

Der Anwender hat eine Eingabe ins Eingabefeld eingegeben.

Ereignisfluss:

1. Das System prüft, ob die Eingabe ein wohlgeformter Ausdruck der Aussagenlogik ist.
2. Das System erstellt die Negation der Eingabeformel.

3. Das System zeigt die Darstellung von der Negation der Eingabeformel als Baumstruktur in dem GUI an.
4. Das System initiiert “Erfüllbarkeit überprüfen” um zu prüfen, ob die Negation der Eingabeformel erfüllbar ist.
5. Wenn dies erfolgreich ist, erstellt das System das Ergebnis, sodass die Eingabeformel nicht allgemeingültig ist. Wenn nicht, erstellt das System das Ergebnis, sodass die Eingabeformel allgemeingültig ist.
6. Das System zeigt die Darstellung des Tableaus an.

Abschlussbedingungen:

Der Anwender hat entweder das Ergebnis oder eine Systemmeldung über eine fehlerhafte Eingabe erhalten.

- **Anwendungsfall: “Erfüllbarkeit überprüfen”**

Akteure:

Anwender (initiiert den Anwendungsfall)

Auslöser:

Anwender drückt auf “Erfüllbarkeit”

Anfangsbedingungen:

Der Anwender hat eine Eingabe ins Eingabefeld eingegeben.

Ereignisfluss:

1. Das System prüft, ob die Eingabe ein wohlgeformter Ausdruck der Aussagenlogik ist.
2. Das System zeigt die Eingabeformel-Darstellung als Baumstruktur in dem GUI an.
3. Das System prüft, ob die Eingabeformel mittels Tableau-Verfahren erfüllbar ist und erstellt das Ergebnis.
4. Das System zeigt die Darstellung des Tableaus an.

Abschlussbedingungen:

Der Anwender hat entweder das Ergebnis oder eine Systemmeldung über eine fehlerhafte Eingabe erhalten.

- **Anwendungsfall: “Schritt für Schritt Lösung anzeigen”**

Akteure:

Anwender (initiiert den Anwendungsfall)

Auslöser:

Anwender drückt auf “Schritt für Schritt Lösung”

Anfangsbedingungen:

Anwender hat “Erfüllbarkeit” oder “Tautologie” gedrückt.

Ereignisfluss:

1. Das System öffnet ein Pop-up Fenster für die Schritt für Schritt Lösung.
2. Der Anwender drückt auf “Nächste Schritt” um den nächsten Schritt der Lösung anzuzeigen.
3. Der Anwender drückt auf “×” Symbol um das Pop-up Fenster zu schließen.
4. Das System schließt das Pop-up Fenster.

Abschlussbedingungen: Pop-up Fenster ist geschlossen. Der Anwender kann die vorherige Formel nochmal prüfen oder eine neue Formel eingeben.

- **Anwendungsfall: “Fehlerhafte Eingabe behandeln”**

Akteure:

System (initiiert den Anwendungsfall)

Auslöser:

Dieser Anwendungsfall erweitert die Anwendungsfälle “Allgemeingültigkeit überprüfe” und “Erfüllbarkeit überprüfen”. Es wird vom System initiiert, sobald die Eingabe nicht ein wohlgeformter Ausdruck der Aussagenlogik ist.

Anfangsbedingungen:

Anwender hat eine fehlerhafte Eingabe eingegeben.

Ereignisfluss:

1. Das System gibt eine Systemmeldung aus.

Abschlussbedingungen:

Der Anwender erhält eine Systemmeldung und kann die Eingabe korrigieren.

3

Grundlagen der Aussagenlogik

In diesem Kapitel werden die Grundlagen der Aussagenlogik behandelt. Die Aussagenlogik ist ein Zweig der formalen Logik, der die Beziehungen zwischen Aussagen und Aussagenverbindungen untersucht. Aussagen sind abstrakte Begriffe, auch Propositionen genannt, die in der Alltagssprache durch Sätze ausgedrückt werden. Dabei kommt es in der Aussagenlogik nicht auf den konkreten Inhalt der Aussagen an, sondern nur auf die Entscheidung, ob eine Aussage wahr oder falsch ist.

Beispiel 3.1

1. Der Mars ist ein Planet.
2. Der Mond ist ein Planet.

drücken zwei verschiedene Aussagen aus, wovon die erste wahr und die zweite falsch ist.

3.1 Syntax der Aussagenlogik

In diesem Abschnitt wird die Syntax von Aussagen exakt spezifiziert, damit ist festgelegt, welche der aus den Grundelementen bildbaren Zeichenfolgen zulässig oder “wohlgeformt” sind und welche nicht.

3.1.1 Alphabet der Aussagenlogik

Definition 3.2 *Alphabet der Aussagenlogik*

Das Alphabet der Aussagenlogik ist einer Menge von erlaubten Zeichen oder Symbolen und besteht aus:

- Aussagenvariablen: $p, q, r, s, t \dots$
- Junktoren:

<i>Negation</i>	\neg
<i>Disjunktion</i>	\vee
<i>Konjunktion</i>	\wedge
<i>Implikation</i>	\rightarrow
<i>Äquivalenz</i>	\leftrightarrow
<i>Exklusives Oder</i>	\oplus
<i>Nor</i>	\downarrow
<i>Nand</i>	\uparrow

- Konstanten: *true* und *false* [Sch13]
- Hilfssymbolen: $(,)$

3.1.2 Syntax aussagenlogischer Formeln

Die folgende Regeln bestimmt welche Zeichenketten, die von dem oberem Alphabet gebildet werden, wohlgeformte Ausdrücke (Formeln) sind, z.B während $(p \rightarrow q)$ eine aussagenlogische Formel ist, ist die $(\wedge p)q\vee$ keine aussagenlogische Formel.

Definition 3.3 Formationsregeln

1. eine Aussagenvariable (atomare Aussage) ist eine Formel.
2. ist A eine aussagenlogische Formel, dann ist auch $\neg A$ eine aussagenlogische Formel.
3. sind A und B aussagenlogische Formeln, dann sind

- (a) $(A \wedge B)$
- (b) $(A \vee B)$
- (c) $(A \rightarrow B)$
- (d) $(A \leftrightarrow B)$
- (e) $(A \oplus B)$
- (f) $(A \downarrow B)$
- (g) $(A \uparrow B)$

ebenfalls aussagenlogische Formeln

4. Ein Ausdruck ist nur dann eine aussagenlogische Formel, wenn er durch Anwendung der obenstehenden Regeln konstruiert werden kann.

Um nachzuweisen, dass bestimmte Zeichenketten keine wohlgeformten Formeln sind, kann man auch beweisen, dass wohlgeformte Formeln bestimmte Eigenschaften haben müssen. Wenn diese Zeichenkette eine dieser Eigenschaften nicht erfüllt, so kann sie auch keine Formel sein.

Bemerkung 3.4 [Dre18]

Solche Eigenschaften von Formeln sind etwa:

- Es gibt ebenso viele öffnende wie schließende Klammern,
- links und rechts von jedem der Junktoren \wedge und \vee steht eine Formel,
- eine Formel endet nie mit einem Junktor.

Beispiel 3.5

Seien p, q aussagenlogische Formeln. Die Zeichenkette $\neg(p \wedge q)$ ist eine Formel, da diese aus den Bildungsregeln (1), (3a) und (2) zusammengesetzt werden können.

Die Zeichenkette $((p \wedge \vee q))$ ist keine Formel, da $\vee q$ (rechts von $p \wedge$) und auch $p \wedge$ (links von $\vee q$) keine Formeln sind. [Dre18]

3.1.3 Bindungskonventionen

Die Klammern um die Ausdrücke sind wichtig, weil durch sie die Reihenfolge der semantischen Auswertung geregelt wird, z.B. Die Formel $(p \vee q) \wedge r$ wird anders ausgewertet als $p \vee (q \wedge r)$. Da viele Klammern Formeln unübersichtlich werden lassen, vereinbart man analog wie in der elementaren Arithmetik “Punktrechnung geht vor Strichrechnung” die folgende Bindungsregeln (von hoch nach niedrig):

$$\begin{array}{c} \neg \\ \wedge, \uparrow \\ \vee, \downarrow \\ \rightarrow \\ \leftrightarrow, \oplus \end{array}$$

3.1.4 Baum-Notation von Formeln

Es ist oft vorteilhaft, sich die Struktur einer Formel A (wie sie durch ihre Unterformeln aufgebaut ist) als einen Syntax-Baum mit Wurzel vorzustellen, dessen Blätter mit den atomaren Aussagen von A und dessen innere Knoten mit geeigneten Junktoren markiert sind. Dabei stimmt die Stelligkeit des markierenden Junktors mit der Anzahl der Kinder überein.

Definition 3.6 *Formeln als Bäume [BA12]*

Eine Formel in der Aussagenlogik ist ein rekursiv definierter Baum:

1. Eine Formel ist ein Blatt, das durch eine atomare Aussage gekennzeichnet ist.
2. Eine Formel ist ein mit einer \neg gekennzeichneteter Knoten mit einem einzelnen Kind, das eine Formel ist.
3. Eine Formel ist ein durch einen der binären Operatoren beschrifteter Knoten mit zwei Kindern, die beide Formeln sind.

Abbildung 3.1 zeigt Baum-Notation von $p \rightarrow p \leftrightarrow \neg p \rightarrow \neg q$:

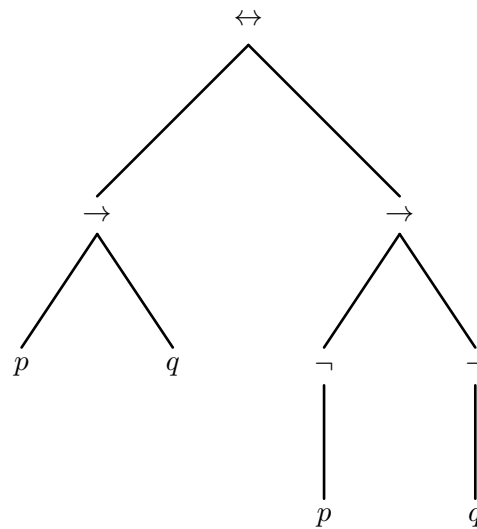


Abb. 3.1. Formel $p \rightarrow p \leftrightarrow \neg p \rightarrow \neg q$ ist ein Baum

Bemerkung 3.7

So wie man Ausdrücke als Strings schreibt (lineare Folgen von Symbolen), kann man Formeln als Strings schreiben. Die einer Formel zugeordnete Zeichenfolge kann durch eine Inorder-Traversierung des Baums erhalten werden.

3.1.5 Operator-Notation

Die Bücher über mathematische Logik benutzen eine stark variierende Notation für die Booleschen Operatoren. Außerdem erscheinen die Operatoren in Programmiersprachen mit einer anderen Notation als es in Mathematikbüchern verwendet wird. Folgende Tabelle zeigt einige dieser alternativen Notationen.

Junktor	Literatur	Java
\neg	\sim	!
\wedge	$\&$	$\&, \&\&$
\vee		$, $
\rightarrow	\supset, \Rightarrow	
\leftrightarrow	\equiv, \Leftrightarrow	
\oplus	\neq	\wedge

Tab. 3.1. Alternative Notationen [BA12]

3.1.6 Eine formale Grammatik für Formeln

Dieser Unterabschnitt setzt Vertrautheit mit formalen Grammatiken voraus. Anstatt Formeln als Bäume zu definieren, können sie als Strings, die über einer kontextfreien formalen Grammatik generiert werden, definiert werden.

Definition 3.8 *Kontextfreien Grammatik für Formel*

Formeln in der Aussagenlogik werden aus der kontextfreien Grammatik abgeleitet, deren Terminals sind [BA12]:

- Eine unbegrenzte Menge von Symbolen \mathcal{P} , die atomare Propositionen heißen.
- Die Booleschen Operatoren in Definition 3.3.

Die Produktionen der Grammatik sind:

$$\begin{aligned}
 fml &::= p \text{ für jedes } p \in \mathcal{P} \\
 fml &::= \neg fml \\
 fml &::= fml \text{ op } fml \\
 op &::= \vee \mid \wedge \mid \rightarrow \mid \leftrightarrow \mid \oplus \mid \uparrow \mid \downarrow
 \end{aligned}$$

3.2 Erfüllbarkeit und Tautologien

Grundsätzlich kann man Formeln danach klassifizieren, ob überhaupt passende Belegungen existieren, die sie wahr oder falsch machen.

Definition 3.9

Eine Formel A heißt:

- *erfüllbar*, falls sie unter wenigstens einer Belegung wahr ist.
- *Tautologie*, falls sie für jede passende Belegung wahr ist.

- *unerfüllbar*, falls es keine passende Belegung wahr gibt.

Satz 3.10

Eine Formel A ist gültig genau dann wenn $\neg A$ unerfüllbar ist.

3.3 Semantisches Tableau

Das Prinzip hinter semantischen Tableaus ist sehr einfach: Man sucht nach einem Modell, das die Interpretation erfüllt, indem man die Formel in mehreren Mengen von Literalen zerlegt. Eine Menge von Literalen ist erfüllbar, wenn die Menge kein komplementäres Literalpaar enthält. Die Formel ist erfüllbar, wenn eine dieser Mengen erfüllbar ist. Die Formel ist unerfüllbar, wenn keine Mengen erfüllbar sind, d.h. seine Negation eine Tautologie ist.

Definition 3.11 *Literal und komplementäres Literalpaar [BA12]*

Ein Literal ist ein Atom oder die Negation eines Atoms. Ein Atom ist ein positives Literal und die Negation eines Atoms ist ein negatives Literal. Für jedes Atom p ist $\{p, \neg p\}$ ein komplementäres Literalpaar.

Für jede Formel A ist $\{A, \neg A\}$ ein komplementäres Formelpaar. A ist das Komplement von $\neg A$ und $\neg A$ ist das Komplement von A .

Beispiel 3.12

In der Menge der Literale $\{\neg p, q, r, \neg r\}$ sind q und r positive Literale, während $\neg p$ und $\neg r$ negative Literale sind. Die Menge enthält das komplementäre Literalpaar $\{r, \neg r\}$.

In der Methode der semantischen Tableaus bezeichnen Mengen von Formeln Knoten eines Baumes, wobei jeder Pfad im Baum die Formeln darstellen, die in einer möglichen Interpretation erfüllt sein müssen.

Definition 3.13 *Konstruktion von semantischen Tableaus*

- Ein aussagenlogisches Tableau ist ein markierter Baum. Jeder Knoten ist mit einer Menge von Formeln markiert.
- Die Eingabeformel ist an der Wurzel.
- Jeder Knoten hat einen oder zwei untergeordnete Knoten, abhängig davon, wie eine Formel, die den Knoten kennzeichnet, zerlegt wird. Dadurch, dass ein Knoten mit α -Formel gekennzeichnet wird, hat er nur einen untergeordneten Knoten, ansonsten hat er zwei Knoten. Die Klassifizierung von α - und β -Formeln wird in der Tabelle 3.2 definiert.

- Die Blätter sind durch eine Menge von Literalen markiert.
- Ein Blatt, das durch eine Menge von Literalen markiert ist, die ein komplementäres Paar von Literalen enthält, ist mit geschlossen (\times) markiert
- Ein Blatt, das durch eine Menge von Literalen markiert ist, die kein komplementäres Paar von Literalen enthält, ist mit offen (\odot) markiert.
- Ein Tableau ist geschlossen, wenn alle seine Blätter als geschlossen markiert sind. Ansonsten (wenn ein Blatt als offen markiert ist), ist es offen. [BA12]

Abbildung 3.2 zeigt semantische Tableaus für die Formeln:

$$A = p \wedge (\neg q \vee \neg p) \text{ und } B = (p \vee q) \wedge (\neg p \wedge \neg q).$$

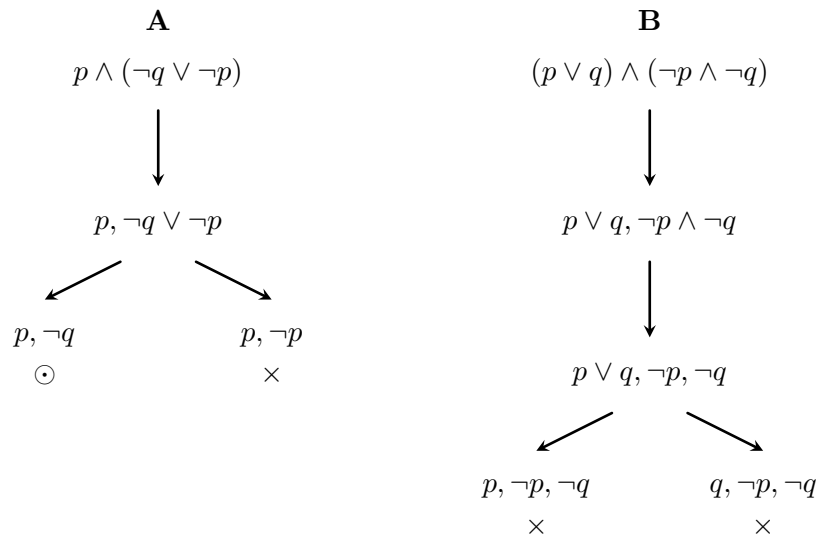
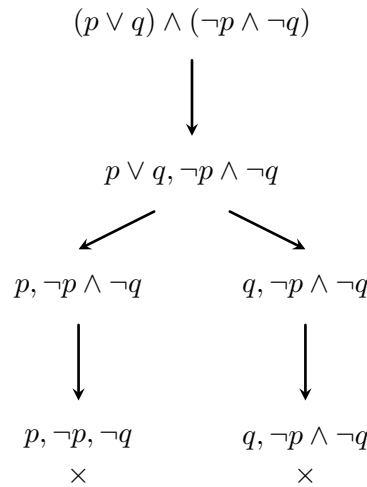


Abb. 3.2. Semantische Tableaus für $A = p \wedge (\neg q \vee \neg p)$ und $B = (p \vee q) \wedge (\neg p \wedge \neg q)$

Die Tableau-Konstruktion ist nicht eindeutig. Abbildung 3.3 ist ein weiteres Tableau für die Formel B . Die Unterschiede zwischen dem Tableau in der Abbildung 3.2 und in der Abbildung 3.3 sind, dass in dem ersten Tableau die Belegung für $\neg p \wedge \neg q$ gesucht werden, bevor die für $p \vee q$ gesucht wird. Das erste Tableau enthält weniger Knoten, was zeigt, dass *Konjunktionen vor Disjunktionen* vorzuziehen sind.

Eine übersichtliche Darstellung der Regeln für die Erstellung eines semantischen Tableaus kann gegeben werden, wenn *Formeln nach ihrem Hauptoperator klassifiziert* werden (Tabelle 3.2). Wenn *die Formel eine Negation ist, berücksichtigt die Klassifizierung sowohl die Negation als auch den Hauptoperator*. α -Formeln sind konjunktiv und nur erfüllbar, wenn beide Teilformeln α_1 und α_2 erfüllt sind, während β -Formeln disjunktiv sind und auch dann erfüllt sind, wenn nur eine der Teilformeln β_1 oder β_2 erfüllbar ist.

**Abb. 3.3.** Ein weiteres Tableau für B

Nr.	α	α_1	α_2	Nr.	β	β_1	β_2
a1	$\neg\neg A_1$	A_1		b1	$\neg(B_1 \wedge B_2)$	$\neg B_1$	$\neg B_2$
a2	$A_1 \wedge A_2$	A_1	A_2	b2	$B_1 \vee B_2$	B_1	B_2
a3	$\neg(A_1 \vee A_2)$	$\neg A_1$	$\neg A_2$	b3	$(B_1 \rightarrow B_2)$	$\neg B_1$	B_2
a4	$\neg(A_1 \rightarrow A_2)$	A_1	$\neg A_2$	b4	$B_1 \uparrow B_2$	$\neg B_1$	$\neg B_2$
a5	$\neg(A_1 \uparrow A_2)$	A_1	A_2	b5	$\neg(B_1 \downarrow B_2)$	B_1	B_2
a6	$A_1 \downarrow A_2$	$\neg A_1$	$\neg A_2$	b6	$\neg(B_1 \leftrightarrow B_2)$	$\neg(B_1 \rightarrow B_2)$	$\neg(B_2 \rightarrow B_1)$
a7	$A_1 \leftrightarrow A_2$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$	b7	$B_1 \oplus B_2$	$\neg(B_1 \rightarrow B_2)$	$\neg(B_2 \rightarrow B_1)$
a8	$\neg(A_1 \oplus A_2)$	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_1$				

Tab. 3.2. Klassifizierung von α - und β -Formeln [BA12]**Beispiel 3.14**

Die Formel $p \wedge q$ wird als α -Formel klassifiziert, weil sie genau dann gilt, wenn sowohl p als auch q wahr sind. Die Formel $\neg(p \wedge q)$ wird als β -Formel klassifiziert. Es ist logisch äquivalent zu $\neg p \vee \neg q$ und ist genau dann wahr, wenn entweder $\neg p$ wahr oder $\neg q$ wahr ist.

Der Algorithmus der Tableau-Konstruktion kann wie folgt formuliert werden:

Algorithmus 3.15 Konstruktion eines semantischen Tableaus [Sch17]

Eingabe: Aussagenlogische Formel A .

Ausgabe: Semantisches Tableau mit markierten Blättern

- 1 **function** Tableau(A)
- 2 Konstruiere einen Baum mit Formel A als Wurzel.
- 3 **while** es gibt unmarkierte Blätter **do**
- 4 Wähle unmarkiertes Blatt l
- 5 **if** Blatt l enthält nur Literale **then**

```
6      if Blatt  $l$  enthält komplementäre Literale oder Konstante false then
7          Markiere Blatt  $l$  als geschlossen mit  $\times$ 
8      else
9          Markiere Blatt  $l$  als offen mit  $\odot$ 
10     end if
11 else if Blatt  $l$  enthält  $\alpha$ -Formel then
12     Wähle  $\alpha$ -Formel  $\alpha$ 
13     Ermittle Formeln  $\alpha_1$  und  $\alpha_2$  nach Tabelle 3.2
14     Erzeuge Kind  $l'$  von  $l$ 
15     Beschrifte  $l'$  mit Formeln aus  $l$ 
16     Ersetze dabei  $\alpha$  in  $l'$  durch Formeln  $\alpha_1$  und  $\alpha_2$ 
17 else
18     Wähle  $\beta$ -Formel  $\beta$ 
19     Ermittle Formeln  $\beta_1$  und  $\beta_2$  nach Tabelle 3.2
20     Erzeuge zwei Kinder  $l_1$  und  $l_2$  von  $l$ 
21     Beschrifte  $l_1$  mit Formeln aus  $l$ 
22     Ersetze dabei  $\beta$  in  $l_1$  durch  $\beta_1$ 
23     Beschrifte  $l_2$  mit Formeln aus  $l$ 
24     Ersetze dabei  $\beta$  in  $l_2$  durch  $\beta_2$ 
25 end if
26 end while
27 return Semantisches Tableau
28 end function
```

4

Frameworks

Dieses Kapitel befasst sich mit den Frameworks für die Umsetzung:

- Antlr4 (für die Analyse der Eingabe)
- Bootstrap (für die Gestaltung der Webanwendung)
- Google Charts (für die Darstellung des Tableaus)
- Jasmine (für das automatisierte Testen)

4.1 Antlr4

Um die syntaktische Richtigkeit der Benutzereingabeformel zu prüfen und die Formel in einem Baum-Struktur darstellen zu können (/LF10/), wird ANTLR verwendet.

ANTLR ist ein leistungsfähiger Parser-Generator, mit dem man strukturierte Text- oder Binärdateien lesen, verarbeiten, ausführen oder übersetzen kann. Es wird häufig in der Wissenschaft und der Industrie verwendet, um alle möglichen Sprachen, Werkzeuge und Frameworks zu erstellen. ANTLR ist kostenlos und als “Open Source” erhältlich.

Die Twitter-Suche verwendet ANTLR für die Abfrageanalyse mit mehr als 2 Milliarden Abfragen pro Tag. Die Sprachen für Hive und Pig und die Data Warehouse- und Analysesysteme für Hadoop verwenden ANTLR. Lex Machina verwendet ANTLR zur Informationsextraktion aus Rechtstexten. Oracle verwendet ANTLR innerhalb der SQL Developer IDE und seiner Migrationstools. Die NetBeans-IDE analysiert C++ mit ANTLR. Die HQL-Sprache im objektrelationalen Hibernate-Mapping-Framework wird mit ANTLR erstellt. [Par12]

Aus einer formellen Sprachbeschreibung, die als Grammatik bezeichnet wird, generiert ANTLR einen Parser für diese Sprache. Ein Parser liest den vom Lexer in einzelne Token zerteilten Strom ein, prüft diese auf syntaktische Richtigkeit und erstellt daraus Symbol-Gruppen mit hierarchischer Struktur. Die Struktur ist oft ein Baum und

wird dann abstrakter Syntaxbaum (AST) oder Parse-tree genannt. ANTLR generiert außerdem automatisch Tree-Walker, mit denen Sie die Knoten dieser Bäume aufrufen können, um anwendungsspezifischen Code auszuführen.

Um einen AST zu bekommen, muss man:

- Eine Lexer- und Parsergrammatik definieren.
- ANTLR aufrufen: Es generiert einen Lexer und einen Parser in einer Zielsprache (z. B. Java, Python, JavaScript)
- Den generierten Lexer und Parser verwenden: Rufe Lexer und Parser, dann übergebe den Code. Lexer und Parser werden den Code erkennen und einen AST zurückgeben

4.1.1 Hauptteilen von ANTLR

ANTLR besteht eigentlich aus zwei Hauptteilen: dem Werkzeug (Antlr-Tool), mit dem der Lexer und Parser erzeugt wird und der Laufzeit, die für die Ausführung benötigt wird. Das Antlr-Tool ist immer gleich, unabhängig davon, auf welche Sprache abgezielt wird. Es ist ein Java-Programm, das nur für Entwickler benötigt wird. Die Laufzeit (Runtime) ist für jede Sprache unterschiedlich und muss sowohl dem Entwickler als auch dem Benutzer zur Verfügung stehen. Die Runtime kann auf der Seite <http://www.antlr.org/download/> heruntergeladen werden.

Die einzige Voraussetzung für das Antlr-Tool ist, dass mindestens Java 1.7 installiert werden muss. Um dieses Antlr-Tool zu installieren, muss die neueste Version von der offiziellen Website, die im Moment <http://www.antlr.org/download/antlr-4.7.1-complete.jar> ist, heruntergeladen werden. Die Ausführliche Anleitung zur Installation wird auf der Seite <https://github.com/antlr/antlr4/blob/master/doc/getting-started.md> beschrieben.

Wenn man ANTLR verwendet, beginnt man mit dem Schreiben einer Grammatik, einer Datei mit der Erweiterung *.g4*. Diese Grammatik erhält die Regeln der Sprache, die analysiert wird. Dann verwendet man das Antlr-Tool, um die Dateien, die man in seinem Projekt wirklich braucht, zu erstellen, wie zB. *antlr4 <Optionen> <Grammatik-Datei-Name.g4>*

Es gibt einige wichtige Optionen, die man bei der Ausführung von Antlr-Tool angeben kann. Man kann die Zielsprache angeben, um einen Parser in Python oder JavaScript oder einem anderen Ziel als Java (das ist das Standardziel) zu erstellen. ANTLR bietet zwei Tree-Walking-Mechanismen in seiner Laufzeitbibliothek: eine Parser-Tree-Listener-Schnittstelle und eine Parser-Tree-Visitor-Schnittstelle [Par12]. Standardmäßig wird nur der Listener generiert. Um den Visitor zu erstellen, kann man die Befehlszeilenoption *-visitor* verwenden. Eine vollständige Liste der Optionen für das Antlr-Tool kann man auf der Seite <https://github.com/antlr/antlr4/blob/master/doc/tool-options.md> finden.

4.1.2 ANTLR JavaScript target

Während der Konfiguration von ANTLR in JavaScript gibt es einige Besonderheiten wie folgt:

- Grammatiken im gleichen Ordner wie die JavaScript-Dateien ablegen.
- Die Datei mit der Grammatik muss den gleichen Namen wie die Grammatik haben, die am Anfang der Datei angegeben werden muss.
- Der entsprechende JavaScript-Parser kann einfach die richtige Option mit dem Antlr-Tool erstellen.

antlr4 -Dlanguage = JavaScript Grammatik-Datei-Name.g4

Damit die Manipulationen erleichtert werden, wird in dieser Arbeit eine *.BAT-Datei antlr4.bat* verwendet. Die Datei erhält die Kommando

```
1  antlr4 -Dlanguage=JavaScript -listener -visitor -encoding UTF-8 FormulaGrammar
   .g4
```

um die Parser, Lexer sowie Listener und Visitor für die *FormulaGrammar.g4* in JavaScript zu generieren.

- ANTLR kann sowohl mit *node.js* als auch im Browser verwendet werden. Für den Browser muss man *webpack* oder *require.js* verwenden um zu vermeiden, dass dutzende Dateien manuell importiert werden müssen.

Im Rahmen diese Bachelorarbeit wird *require.js* gewählt. Das Skript wird von Torben Haase zur Verfügung gestellt und ist nicht Bestandteil der ANTLR JavaScript-Runtime. Dieses Skript kann auf der Seite <https://github.com/antlr/antlr4/blob/master/runtime/JavaScript/src/lib/require.js#L65> heruntergeladen werden. Angenommen hat man im Stammverzeichnis seiner Website sowohl das Verzeichnis “antlr4” als auch ein “lib” -Verzeichnis mit “require.js”, muss man die in denn HTML-Header wie folgt einfügen:

```
1  <script src='lib/require.js'>
2  <script>
3      var antlr4 = require('antlr4/index');
4  </script>
```

- Die generierten Lexer und Parser in JavaScript kann wie folgt ausgeführt werden:

```
1  var antlr4 = require('antlr4');
2  var MyGrammarLexer = require('./MyGrammarLexer').MyGrammarLexer;
3  var MyGrammarParser = require('./MyGrammarParser').MyGrammarParser;
4  var MyGrammarListener = require('./MyGrammarListener').MyGrammarListener;
5
6  var input = "your text to parse here"
7  var chars = new antlr4.InputStream(input);
8  var lexer = new MyGrammarLexer(chars);
9  var tokens = new antlr4.CommonTokenStream(lexer);
10 var parser = new MyGrammarParser(tokens);
11 parser.buildParseTrees = true;
12 var tree = parser.StartRule();
```

Dazu sind *MyGrammarLexer.js*, *MyGrammarParser.js*, *MyGrammarListener.js* und *MyGrammarVisitor.js* die generierten Dateien für die Grammatik “*MyGrammar*”, die eine Regel “*StartRule*” enthält.

- Wenn man den Syntaxbaum mit einem benutzerdefinierten Listener besuchen möchte, kann man einen benutzerdefinierten Listener wie folgt erstellen:

```

1  MyGrammarListener = function(ParseTreeListener) {
2    // some code here
3  }
4  // some code here
5  MyGrammarListener.prototype.enterKey = function(ctx) {};
6  MyGrammarListener.prototype.exitKey = function(ctx) {};
7  MyGrammarListener.prototype.enterValue = function(ctx) {};
8  MyGrammarListener.prototype.exitValue = function(ctx) {};

```

Um benutzerdefiniertes Verhalten bereitzustellen, kann man eine Klasse wie folgt erstellen:

```

1  var KeyPrinter = function() {
2    MyGrammarListener.call(this); // inherit default listener
3    return this;
4  };
5
6  // continue inheriting default listener
7  KeyPrinter.prototype = Object.create(MyGrammarListener.prototype);
8  KeyPrinter.prototype.constructor = KeyPrinter;
9
10 // override default listener behavior
11 KeyPrinter.prototype.exitKey = function(ctx) {
12   console.log("Oh, a key!");
13 };

```

Um diesen Listener auszuführen, fügt man dem obigen Code einfach die folgenden Zeilen hinzu:

```

1  ...
2  tree = parser.StartRule() // only repeated here for reference
3  var printer = new KeyPrinter();
4  antlr4.tree.ParseTreeWalker.DEFAULT.walk(printer, tree);

```

Die Ausführliche Anleitung zur JavaScript Konfiguration kann man auf der Seite <https://github.com/antlr/antlr4/blob/master/doc/javascript-target.md> lesen.

4.2 Bootstrap

Immer mehr Smartphones, Tablets und andere Mobilgeräte werden genutzt, um im Internet zu surfen. Weltweit werden 68 Millionen Google-Suchanfragen pro Stunde auf Mobilgeräten durchgeführt. Laut StatCounter waren im Juli 2018 52,95% des weltweiten Internetverkehrs auf einem Mobilgerät. Weitere 3,94% kamen von Tablets [Sta18]. Das bedeutet, dass Mobile Nutzer einen großen Einfluss haben, mit dem man rechnen muss um den langfristigen Erfolg sicherzustellen. Es gibt zwei Optionen zum Erstellen einer für Mobilgeräte optimierten Website. Man kann eine separate mobile Website oder eine Responsive-Website entwickeln.

4.2.1 Eigenständige mobile Website (Separate mobile Website)

Separate mobile Websites sind Websites, die speziell für Mobilgeräte entwickelt wurden. Sie leben häufig unter einer separaten URL (z.B. *m.site.de*) und unterscheiden sich

von der vollständigen Website. Sie enthalten Funktionen oder Inhalte, die für Mobilgeräte als geeignet erachtet wurden. Häufig sind dies nur einige der auf dem Desktop verfügbaren Komponenten. Eine separate mobile Website bietet Differenzierung von mobilen Inhalten und erstellt ein vollständig mobiles Benutzererlebnis.

Das größte Problem mit separaten mobilen Websites besteht darin, dass man zwei separate Websites erstellen und verwalten muss. Wenn man Änderungen an der einen Website vornehmen muss, muss man dieselben Änderungen auf der anderen Website wiederholen. Dies könnte mehr Zeit und Geld kosten. Auch wenn man es auf einer separaten Domain platziert, muss man für eine neue Domain und das Hosting bezahlen. Die Umleitung kann die Ladezeit beeinflussen, was die Absprungrate erhöhen kann, da Besucher ungeduldig sein können.

4.2.2 Sich anpassendes Design (Responsive-Design)

Der Begriff Responsive-Design, zuerst von Ethan Marcote im Jahr 2010 geprägt, beschreibt eine Entwicklungstechnik, bei der das Design einer Website automatisch an die Größe der Benutzerbildschirme angepasst wird. Daher kann derselbe Inhalt in einem dreispaltigen Format auf einem Desktop, einem zweispaltigen Format auf einem Tablet und einem einspaltigen Format auf einem Smartphone angezeigt werden. Kurztipp: Man kann feststellen, ob eine Website “responsive” ist, indem man das Browserfenster manuell vergrößert oder verkleinert.

Responsive-Design liefert für jede Seite unabhängig vom Gerät den gleichen Code über eine einzige URL an den Browser. Man muss nicht mehr zwei Versionen einer Website erstellen: eine für Desktop-Computer und eine für mobile Geräte. Da es sich nur um eine Website handelt, ist eine Responsive-Website einfacher und kostengünstiger zu warten. Alle Änderungen, die man vornimmt, sind sowohl in der mobilen Version als auch in der Desktop Version sichtbar. Außerdem sind Responsive-Websites oft einfacher zu implementieren und weniger kompliziert in Bezug auf die Konfiguration für Suchmaschinen.

Es gibt jedoch auch Nachteile, die mit Responsive-Design einhergehen. Da eine Responsive-Website als eine einzelne Entität codiert ist (im Gegensatz zu einer Desktop-Website und dann einer separaten mobilen Website), müssen alle Seitenressourcen und Code für jeden Besuch des Benutzers heruntergeladen werden, unabhängig von der Bildschirmgröße oder Gerät, das sie besuchen. In mobilen Versionen von Responsive-Website werden häufig bestimmte Elemente nicht angezeigt, um Webseiten oder Abschnitte benutzerfreundlicher zu machen. Diese Elemente müssen jedoch “unsichtbar” geladen werden. Dies bedeutet, dass die Webseiten mit dem Responsive-Design wahrscheinlich langsamer geladen werden.

4.2.3 Mobile-first

Außerdem gibt es noch die Begriffe “Desktop-First” und “Mobile-First”. Während Responsive-Design eine Entwicklungstechnik ist, sind Desktop-First und Mobile-First die Design-Strategien [Gon18].

Desktop-First ist ein Konzept im Responsive-Design bei dem als erstes die Website für die Desktop-Darstellung entwickelt wird. Für kleinere Displays wird die Seite im Nachhinein angepasst [kul18].

Mobile-First, eine Idee von Luke Wroblewski, ist ein Trend in der Website-Entwicklung, bei der das Entwerfen einer Website für Smartphones, Tablets und mobilen Geräte Vorrang vor der Desktop-Version hat. Mit Mobile-First wird ein Webdesigner angesichts der Einschränkungen einer mobilen Plattform (kleiner Bildschirm, langsamere Prozessoren) eine Website erstellen und dann die Website entweder für die Desktop Nutzung kopieren oder verbessern.

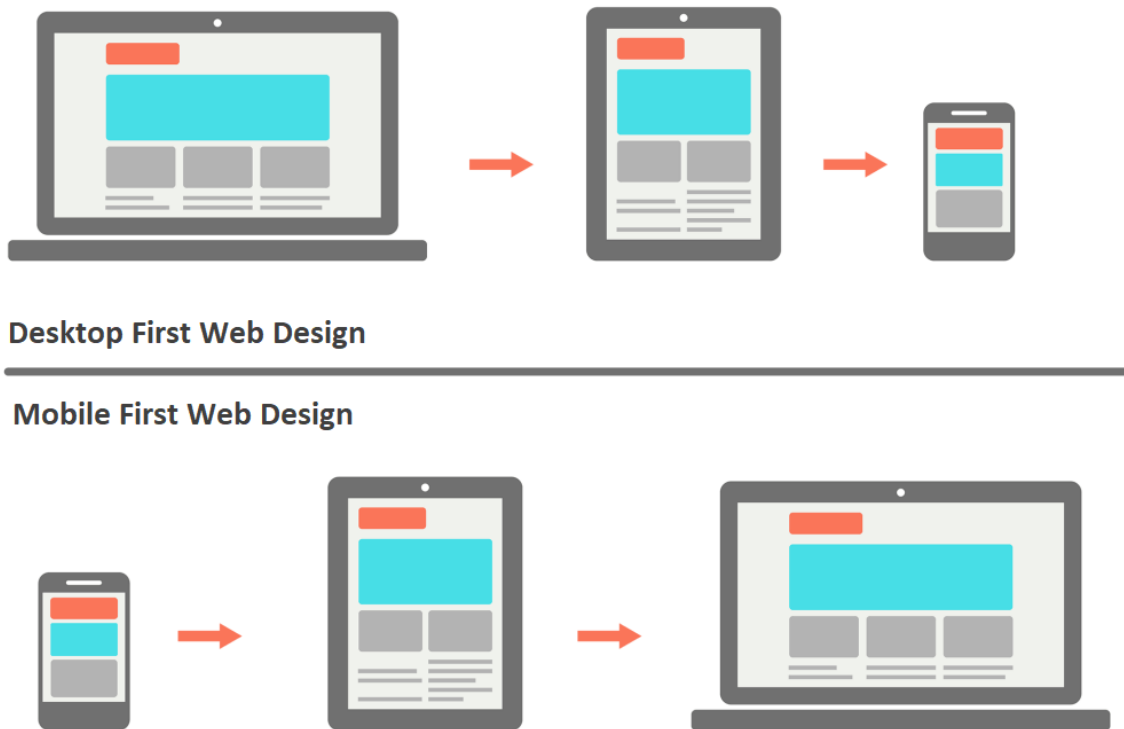


Abb. 4.1. Desktop-First und Mobile-First

Quelle: <http://fredericgonzalo.com/en/2017/03/01/understanding-the-difference-between-mobile-first-adaptive-and-responsive-design/>

4.2.4 Bootstrap

Aufgrund der Unterstützung für die Informatik Studierenden (/LE10/und /LE40/), die den Laptop fast täglich verwenden müssen, den Wartungsaufwand und die Implementierungskomplexität, wird eine Responsive-Website mit “Laptop-First” Strategie (Bildschirmgröße ≥ 768 px) gewählt. Um eine Responsive-Website schneller und einfacher gestalten zu können, wird Bootstrap verwendet.

Bootstrap ist ein beliebtes, kostenloses HTML-, CSS- und JavaScript-Framework zum Entwickeln von Responsive-Websites mit Priorisierung von Mobilgeräten. Das Framework enthält Responsive-CSS- und -HTML-Vorlagen für Schaltflächen, Tabellen, Navigation, Bildkarussells und andere Elemente, die man auf der Webseite verwenden kann. Es sind einige optionale JavaScript-Plugins verfügbar, die selbst Entwick-

lern mit lediglich grundlegenden Codierungskenntnissen das Entwickeln großartiger Responsive-Websites ermöglichen. Bootstrap ist mit allen modernen Browsern kompatibel wie (Chrome), Firefox, Internet Explorer, Safari und Opera [Boo18b].

	Chrome	Firefox	Internet Explorer	Opera	Safari
Mac	✓ Supported	✓ Supported	N/A	✓ Supported	✓ Supported
Windows	✓ Supported	✓ Supported	✓ Supported	✓ Supported	✗ Not supported

	Chrome	Firefox	Safari
Android	✓ Supported	✓ Supported	N/A
iOS	✓ Supported	✓ Supported	✓ Supported

Abb. 4.2. Unterstützte Browser von Bootstrap

Quelle: <https://getbootstrap.com/docs/3.3/getting-started/#support>

Bootstrap wurde 2010 von Twitter unter dem Namen „Twitter Bootstrap“ entwickelt, mit dem Ziel eine einheitliche Bibliothek für die Gestaltung von Weboberflächen zu schaffen. Das Problem war damals, dass für die Designentwicklung bei Twitter viele verschiedene Bibliotheken verwendet wurden. Das führte zu Inkonsistenzen und einem großen Wartungsaufwand. Bootstrap sollte eine gemeinsame Basis schaffen, mit der alle Mitarbeiter arbeiten konnten, um schnell und einfach Websites zu gestalten. Im August 2011 entschloss sich Twitter dazu, das Bootstrap Framework als Open Source Projekt zu veröffentlichen. Damit war der Siegeszug dieses ausgezeichneten und leicht zu bedienenden Frontend-Frameworks zur Webdesign-Gestaltung nicht mehr aufzuhalten [Boo18a].

Um Bootstrap zu verwenden, muss man HTML und CSS nicht gut kennen. Es ist ein Vorteil, wenn man ein Backend-Entwickler ist und einige UI-Änderungen vornehmen kann.

Bootstrap-Bibliothek kann als ein ZIP-Archiv von der Bootstrap-Webseite <http://getbootstrap.com/docs/3.3/getting-started/#download> heruntergeladen werden. Dieses Archiv enthält bereits fast alle benötigten, in das eigene Projekt einzubindenden Dateien, wie eine Stylesheetdatei mit allen Komponenten, eine JavaScript-Datei mit allen Plugins und auch eine benötigte Icon-Schriftart. Alternativ gibt es auf GitHub noch ein vollständiges, deutlich umfangreicheres ZIP-Archiv für Entwickler herunterzuladen, welches auch Beispiele für typische Webseiten zur bequemen Verwendung als Ausgangsdatei und vieles weitere enthält.

Die Dateien im ZIP-Archiv müssen in das eigene HTML-Dokument/Projekt eingebunden werden. Soll auch mit JavaScript-Komponenten gearbeitet werden, so muss die JavaScript-Datei zusammen mit der jQuery-Bibliothek ebenfalls im HTML-Dokument referenziert werden. Möchte man angepasste Einstellungen für Stil und JavaScript-Funktionalität, besteht die Möglichkeit, fast alle Elemente von Bootstrap auf der Website selbst zu verändern und ein angepasstes Paket herunterzuladen. Schließlich kann man Bootstrap auch lokal, seinen Bedürfnissen entsprechend, vom Standard abweichend kompilieren.

Das folgende Beispiel verdeutlicht die Funktionsweise. Der HTML-Quellcode definiert ein einfaches Suchformular sowie eine Ergebnisliste in Form einer Tabelle. Die Sei-

te besteht aus regulären, semantisch verwendeten HTML5-Elementen sowie einigen zusätzlichen CSS-Klassenangaben entsprechend der Bootstrap-Dokumentation [Wik18a].

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Bootstrap Beispiel</title>
5     <!-- Einbinden des Bootstrap-Stylesheets -->
6     <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/
bootstrap.min.css">
7     <!-- optional: Einbinden der jQuery-Bibliothek -->
8     <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-1.12.4.min.js"></script
>
9     <!-- optional: Einbinden der Bootstrap-JavaScript-Plugins -->
10    <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"></
script>
11  </head>
12  <body>
13    <section class="container">
14      <h1>Suche</h1>
15      <p>Beispiel für ein einfaches Suchformular.</p>
16      <!-- Suchformular mit Eingabefeld und Button -->
17      <form class="well form-search">
18        <input type="text" class="input-medium search-query"/>
19        <button type="submit" class="btn btn-primary">Search</button>
20      </form>
21      <h2>Ergebnisse</h2>
22      <!-- Tabelle mit abwechselnder Zellenhintergrundfarbe und Außenrahmen -->
23      <table class="table table-striped table-bordered">
24        [...]
25      </table>
26    </section>
27  </body>
28 </html>

```

Listing 4.1. Bootstrap Beispiel

Grid System

Das Grid-System ist eines der wichtigsten Konzepte in Bootstrap, das beschreibt, wie man Komponenten auf der Oberfläche positionieren und wie man eine Responsive-Website implementieren kann. Im Grid-System ist das Layout der Seite in verschiedene Zeilen aufgeteilt. Jede Zeile hat maximal 12 Spalten (möglicherweise weniger). Basierend auf der Breite des Anzeigegerätetyps kann man verschiedene CSS-Klassen verwenden, um die Anzahl der Anzeigespalten anzupassen. Zum Beispiel hat man eine Webseite, die das Grid Layout System wie folgt verwendet:

Width: 8.33%	Width: 8.33%	Width: 8.33%	Width: 8.33%	Width: 8.33%	Width: 8.33%	Width: 8.33%	Width: 8.33%	Width: 8.33%	Width: 8.33%	Width: 8.33%	Width: 8.33%
Width: 66.67%								Width: 33.33%			
Width: 33.33%				Width: 33.33%				Width: 33.33%			
Width: 50%						Width: 50%					

Abb. 4.3. Beispiel Grid System

Die Abbildung 4.3 hat vier verschiedene Zeilen, wobei jede Zeile eine andere Anzahl von Spalten hat, beispielsweise die erste Zeile hat 12 Spalten. Die CSS-Klasse, die auf die Spalte angewendet wird, ist in fünf verschiedene Kategorien unterteilt (Tabelle 4.1).

	Extra klein < 576px	Klein ≥ 576px	Medium ≥ 768px	Groß ≥ 992px	Extragroß ≥ 1200px
Maximale Behälterbreite	Keine (automatisch)	540px	120px	960px	1140px
Klassenpräfix	.col-	.col-sm-	.col-md-	.col-lg-	.col-xl-
Anzahl der Spalten	12				
Stegbreite	30px(15px auf jeder Seite einer Spalte)				
Nestbar	Ja				
Spaltenbestellung	Ja				

Tab. 4.1. Rasteroptionen

Quelle: <https://getbootstrap.com/docs/4.1/layout/grid/>

Wenn es nur 1 Block `<div class="col-md"> </div>` und keine Spaltennummer gibt, ist der Standardwert 12 Spalten (voller Block von Containern). Wenn man 3 Blöcke `<div class="col-md"> </div>` hat, werden die Blöcke automatisch ebenso wie `<div class="col-md-4"> </div>` unterteilt. Die Bildschirme, die größer als den verwendeten Bildschirm sind, ändern sich nicht. Die kleineren Bildschirme werden automatisch auf `col-12` (eine Spalte pro Zeile) umgeschaltet, wenn keine andere Anpassung vorgenommen wird. Das folgende Beispiel verdeutlicht die Funktionsweise des Grid System.

Zuerst fügt man dieses Meta-Tag dem Head-Tag hinzu.

```
1 <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Und das Body-Tag ist wie folgt:

```
1 <div class="container" style="background-color: grey; margin-bottom: 50px;">
2   <div class="row">
3     <div class="col-md-4 col-sm-6">
4       <div class="block"></div>
5     </div>
6     <div class="col-md-8 col-sm-6">
7       <div class="block"></div>
8     </div>
9   </div>
10 </div>
```

Listing 4.2. Beispiel Grid System

Ergebnisse des Laufs des obigen Codes:

- Wenn die Bildschirmbreite größer als 768px ist, wird jede Zeile in zwei Spalten aufgeteilt: Eine Spalte belegt 33,33 % der Bildschirmbreite und eine Spalte belegt 66,67 % der Bildschirmbreite.
- Wenn die Bildschirmbreite weniger als 768px ist, wird jede Zeile in zwei Spalten aufgeteilt: Jede Spalte belegt 50% der Bildschirmbreite.
- Wenn die Bildschirmbreite weniger als 576px ist, hat jede Zeile nur eine Spalte.

Die Dokumentation von Bootstrap ist essentiell. Bootstrap hat eine umfangreiche Dokumentation inklusive Demos, die auch für Anfänger in dem Bereich leicht nachvollziehbar sind und bis hin zu den komplexesten Elementen alles für den Nutzer zugänglich macht. Unter <http://getbootstrap.com/getting-started/#template> kann man eine grundlegende Vorlage und eine Reihe von Beispielen für unterschiedliche Bedürfnisse

<http://getbootstrap.com/getting-started/#examples> herunterladen. Man kann einfach das Bootstrap-Repository herunterladen, in den Ordner *docs / examples* gehen, das gewünschte Beispiel kopieren / einfügen und daran arbeiten.

4.3 Google chart tools

Um den Tableau zu visualisieren, braucht man ein Werkzeug, das einfach zu bedienen und kostenlos ist. Mit dieser Anforderungen ist Google Charts ein guter Kandidat.

Google chart tools [Cha18](Google Charts, unterscheidet sich von dem Google Chart API) ist ein interaktiver Webservice [Wik18b], mit dem Benutzer ihre Daten auf ihrer Website über einfache oder attraktive Visualisierungen anzeigen können. Das Werkzeug wird häufig mit einem einfachen JavaScript verwendet, das auf der Webseite eingebettet ist. Mit Google Charts können Benutzer die einfachen Diagramme wie Liniendiagramme bis hin zu komplexen Diagrammen wie Baumdiagramme, erstellen.

Neben dem standardmäßigen Google-Design werden den Nutzern auch zahlreiche Anpassungsoptionen für ihre Diagramme zur Verfügung gestellt. Das Werkzeug ist recht einfach zu bedienen, da Benutzer nur die Anwendung einbetten, die Google Chart-Bibliotheken laden und die zu charternden Daten eingeben müssen. Nach ein paar Anpassungen und der Zuweisung einer ID kann das Diagramm auf der Webseite aktiviert werden.

Google Charts ist nicht nur kostenlos, sondern auch eine benutzerfreundliche Anwendung. Mit ein wenig JavaScript-Kenntnissen kann man komplizierteste Diagramme und Grafiken erstellen. Die Anpassungsoptionen sind auch ein weiterer erwähnenswerter Vorteil. Diagramme können mit Farben, Linien, Überlagerungen, Punkten usw. angepasst und optimiert werden, damit sie sich leicht an die Schnittstelle der Webseite anpassen. Es gibt eine großartige Dokumentation von Google, die unter der Seite <https://developers.google.com/chart/interactive/docs/> nachgeschlagen werden kann.

Gegen die Verwendung von Google Charts spricht zum einen, dass eine Netzwerkverbindung erforderlich ist. Außerdem ist es durch Google den Benutzern nicht gestattet, den Code selbst zu speichern oder zu hosten. Diese Einschränkungen haben jedoch keinen Einfluss auf die Anforderungen der Tableau-Anwendung.

4.3.1 Organigramm

Da die Struktur von Tableau ein Baum ist, eignet sich das Organigramm von Google Charts gut zur Visualisierung eines Tableaus.

Organigramme sind ein Diagramm einer Knotenhierarchie, die häufig verwendet wird, um übergeordnete / untergeordnete Beziehungen in einer Organisation darzustellen. Eine Baumfamilie ist eine Art Organigramm [Goo18b]. Der folgenden HTML-Quellcode definiert ein einfaches Organigramme [Goo18b].

```
1 <html>
```

```

2 <head>
3 <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></
  script>
4 <script type="text/javascript">
5   google.charts.load('current', {packages:["orgchart"]});
6   google.charts.setOnLoadCallback(drawChart);
7
8   function drawChart() {
9     var data = new google.visualization.DataTable();
10    data.addColumn('string', 'Name');
11    data.addColumn('string', 'Manager');
12    data.addColumn('string', 'ToolTip');
13
14    // For each orgchart box, provide the name, manager, and tooltip to show.
15    data.addRows([
16      [{v:'Mike', f:'Mike<div style="color:red; font-style:italic">President</div>
17      ', 'The President'],
18      [{v:'Jim', f:'Jim<div style="color:red; font-style:italic">Vice President</
19      div>'},
20      ['Mike', 'VP'],
21      ['Alice', 'Mike', ''],
22      ['Bob', 'Jim', 'Bob Sponge'],
23      ['Carol', 'Bob', '']
24    ]);
25
26    // Create the chart.
27    var chart = new google.visualization.OrgChart(document.getElementById('
28    chart_div'));
29    // Draw the chart, setting the allowHtml option to true for the tooltips.
30    chart.draw(data, {allowHtml:true});
31  }
32 </script>
33 </head>
34 <body>
35   <div id="chart_div"></div>
36 </body>
37 </html>

```

Listing 4.3. Organigramme Beispiel

Zuerst muss man den Loader selbst laden, was in einem separaten *script*-Tag mit erfolgt:

```

1 src="https://www.gstatic.com/charts/loader.js"

```

Dieses Tag kann sich entweder im “head” oder “body” des Dokuments befinden oder dynamisch in das Dokument eingefügt werden. Nachdem der Loader geladen wurde, kann man *google.charts.load* abrufen. Ab Version 45 kann man *google.charts.load* mehr als einmal abrufen, um weitere Pakete zu laden, aber wenn man dies tut, muss man jedes Mal dieselbe Versionsnummer und dieselbe Spracheinstellung angeben.

```

1 google.charts.load('current', {packages:"orgchar"});

```

Das erste Argument des *google.charts.load* ist der Name oder die Nummer der Version als String. Wenn man *current* angibt, wird dadurch die neueste offizielle Version von Google Charts geladen. Der zweite Parameter des *google.charts.load* ist ein Objekt zum Festlegen von Einstellungen wie Pakete, Sprache, Rückrufen... In dem obigen Beispiel ist *orgchar* der Paketname.

Bevor man eines der von *google.charts.load* geladenen Pakete verwenden kann, muss man warten, bis das Laden beendet ist. Da es einige Zeit dauern kann, bis das Laden beendet ist, muss man eine Rückruffunktion registrieren. Es gibt zwei Möglichkeiten,

dies zu tun. Man gibt entweder eine callback Einstellung in dem *google.charts.load* an oder ruft *setOnLoadCallback* mit einer Funktion (z.B. *drawChart()*) als Argument auf.

4.3.2 DataTable

Alle Diagramme benötigen Daten. Google Chart Tools-Diagramme erfordern das Umbrechen von Daten in eine JavaScript-Klasse namens *google.visualization.DataTable*. Diese Klasse ist in der Google Visualization-Bibliothek definiert, die man zuvor geladen hat.

Die Organigramm erfordert eine *DataTable* mit drei Spalten, wobei jede Zeile einen Knoten im Organigramm darstellt. Hier sind die drei Spalten:

- Spalte 0 : Die Knoten-ID. Es sollte unter allen Knoten *eindeutig* sein und beliebige Zeichen einschließlich Leerzeichen enthalten. Dies wird auf dem Knoten angezeigt. Man kann einen formatierten Wert angeben, der stattdessen im Diagramm angezeigt wird. Der unformatierte Wert wird jedoch weiterhin als ID verwendet.
- Spalte 1(optional): Die ID des übergeordneten Knotens.
- Spalte 2(optional): Tooltip-Text, der angezeigt wird, wenn ein Benutzer den Mauszeiger über diesen Knoten bewegt.

Jeder Knoten kann keinen oder einen übergeordneten Knoten und keinen oder mehrere untergeordnete Knoten haben.

4.3.3 Zeichnen des Diagramms

Der letzte Schritt ist das Zeichnen des Diagramms. Zuerst muss man eine Instanz der Diagrammklasse, die man verwenden möchte, instanziiieren und dann muss man *draw()* aufrufen. Jeder Diagrammtyp basiert auf einer anderen Klasse, die in der Diagrammdokumentation aufgeführt ist, z.B. basiert das Organigramm auf der *google.visualization.OrgChart*.

```
1 var chart = new google.visualization.OrgChart(document.getElementById('chart_div'));
```

Nachdem man seine Daten und Optionen vorbereitet hat, kann man das Diagramm zeichnen. Die Seite muss ein HTML-Element (normalerweise ein *<div>*) enthalten, um das Diagramm zu halten. Man muss dem Diagramm einen Verweis auf dieses Element übergeben, also weist man ihm eine ID zu, mit der man einen Verweis *document.getElementById()* abrufen kann. Alles in diesem Element wird beim Zeichnen durch das Diagramm ersetzt.

Jedes Diagramm unterstützt eine *draw()* Methode [Goo18a], die zwei Werte akzeptiert: ein *DataTable* (oder ein *DataView*) Objekt, das seine Daten enthält, und ein optionales Diagrammoptionsobjekt. Das Optionsobjekt ist nicht erforderlich, und man

kann es ignorieren oder *Null* übergeben, um die Standardoptionen des Diagramms zu verwenden.

Nach dem Aufruf *draw()* wird das Diagramm auf der Seite gezeichnet. Man soll die *draw()* Methode jedes Mal aufrufen, wenn man die Daten oder die Optionen ändern und das Diagramm aktualisieren möchte. Die *draw()* Methode ist asynchron, d.h. sie wird sofort zurückgegeben während die zurückgegebene Instanz jedoch möglicherweise nicht sofort verfügbar ist.

Außerdem gibt es noch viele Einstellungsmöglichkeiten wie Farbe oder Größe der Diagramme sowie die Methoden und Events..., die unter der Seite <https://developers.google.com/chart/interactive/docs/gallery/orgchart> nachgeschlagen werden.

4.4 Jasmine

Um schon während der Entwicklung eine gute Qualität des Quelltextes anzustreben, ist der Einsatz von einem Testing-Framework sinnvoll. Die Wahl fällt auf Jasmine, da es eine einfache Implementierung ermöglicht sowie clientseitig ohne einen Server funktioniert.

Jasmine ist ein verhaltensorientiertes Entwicklungsframework zum Testen von JavaScript-Code. Es hängt nicht von anderen JavaScript-Frameworks ab. Es benötigt kein DOM. Und es hat eine saubere, explizite Syntax, so dass man problemlos Tests schreiben kann [Jas18]. Zusätzlich folgt Jasmine der BDD-Prozedur (Behavior Driven Development), um sicherzustellen, dass jede Zeile der JavaScript-Anweisung ordnungsgemäß Unit-getestet ist. Durch das Befolgen der BDD-Prozedur bietet Jasmine eine kleine Syntax, um die kleinste Einheit der gesamten Anwendung zu testen, anstatt sie als Ganzes zu testen.

Im Folgenden sind die Vorteile der Verwendung von Jasmine gegenüber anderen verfügbaren JavaScript-Test-Frameworks aufgeführt [Poi18].

- Jasmine ist von keinem anderen JavaScript-Framework abhängig.
- Jasmine benötigt kein DOM.
- Die gesamte im Jasmine-Framework verwendete Syntax ist sauber und offensichtlich.
- Jasmine wird stark von Rspec, JS Spec und Jspec beeinflusst.
- Jasmine ist ein Open-Source-Framework und leicht verfügbar in verschiedenen Versionen wie Stand-alone, Ruby-Juwel, Node.js, etc.

4.4.1 Wie benutzt man Jasmin?

Jasmine ist sehr einfach in jeder Art von Entwicklungsmethodik zu implementieren. Alles, was man herunterladen muss, ist die Standalone-Bibliotheksdateien von der offi-

ziellen Website <https://github.com/jasmine/jasmine/releases>. Dieses implementiert man gleich in seiner Anwendung. Nach dem Entpacken der Jasmine-Bibliothek muss man den Entpacker-Ordner in den Unit-Tests-Ordner der erstellten Anwendung einfügen. Der Entpacker-Ordner umfasst:

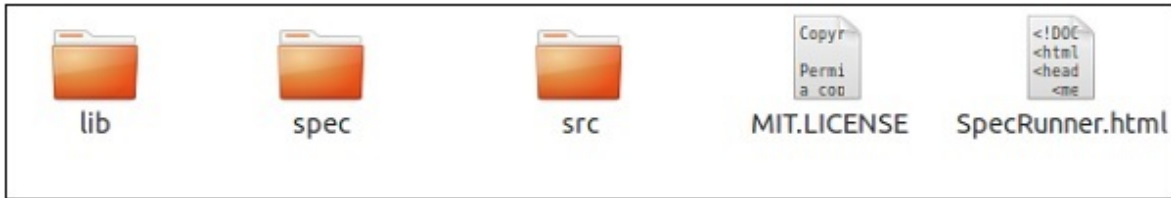


Abb. 4.4. Jasmine Standalone-Bibliotheksdateien

Quelle: https://www.tutorialspoint.com/jasminejs/jasminejs_overview.htm

- *Lib*: enthält Quellcode für das Framework.
- *Spec*: enthält Code für seine Tests
- *Src*: enthält Quellcode für seine Anwendung
- *SpecRunner.html*: ist eine Seite, die man Tests ausführt.

Dieses Paket enthält auch ein kleines Beispiel, das nützlich sein könnte.

4.4.2 Wie man mit Jasmine JS Tests prüft?

Ein einfaches Beispiel für JavaScript-Tests von Jasmine sieht folgendermaßen aus:

```

1 describe("App", function () {
2     describe("foo", function () {
3         it("should return bar", function () {
4             expect(App.foo()).toEqual("bar");
5         });
6     });
7 });
8
9 var App = {
10     foo: function() {
11         return "bar123";
12     }
13 };

```

Im obigen Beispiel wird *App.foo()* durch die Suite “App” mit Matcher *.toEqual* getestet.

Eine Suite stellt eine Reihe von verwandten Tests dar. Jede Suite enthält wiederum eine Reihe von Erwartungen, die die Ergebnisse des Tests - den tatsächlichen Wert - mit dem erwarteten Wert vergleichen. Eine Suite wird durch Aufrufen der Funktion *describe()* definiert. Es benötigt zwei Parameter: den Namen der Suite und die Funktion, die die Aufrufe der Erwartungs-Methoden enthält. Diese werden mit der Methode *it()* definiert. Wie *describe()* akzeptiert *it()* auch einen Namen und einen Funktionsparameter. Der Funktionsparameter *it()* kann Variablen und einen oder mehrere Aufrufe der

expect() enthalten. In Verbindung mit einer Matcher-Funktion führen diese die Aufgabe durch, die Ist- und Erwartungswerte zu vergleichen. Jeder Matcher implementiert einen booleschen Vergleich zwischen dem tatsächlichen Wert und dem erwarteten Wert. Es ist verantwortlich für die Berichterstattung an Jasmine, wenn die Erwartung wahr oder falsch ist. Jasmine wird dann die Spezifikation bestehen oder nicht bestehen.

Jeder Matcher kann eine negative Assertion auswerten, indem er den Call *expect()* mit einem “*not*” vor dem Aufruf des Matcher verkettet. Jasmine bietet eine große Auswahl an Matching-Tools. Die vollständige Liste findet man auf der Seite <https://jasmine.github.io/api/edge/matchers.html>.

Um einer Test-Suite zu helfen, doppelten Konfigurations- und Teardown-Code zu löschen, stellt Jasmine die globalen Funktionen *beforeEach*, *afterEach*, *beforeAll* und *afterAll* bereit. Wie der Name schon sagt, wird die *beforeEach* Funktion einmal vor jeder Spezifikation in der *describe* aufgerufen, in der sie aufgerufen wird und die *afterEach* Funktion wird einmal nach jeder Spezifikation aufgerufen. Die *beforeAll* Funktion wird nur einmal aufgerufen, bevor alle Spezifikationen *describe* ausgeführt werden und die *afterAll* Funktion wird nach Abschluss aller Spezifikationen aufgerufen *beforeAll* und *afterAll* kann verwendet werden, um Testsuiten mit teurem Setup und Teardown zu beschleunigen.

```
1 describe ( "A suite with some common settings" , function(){
2   var foo = 0 ;
3   beforeEach ( Funktion(){
4     foo += 1 ;
5   });
6   afterEach ( Funktion(){
7     foo = 0 ;
8   });
9   beforeAll ( Funktion(){
10    foo = 1 ;
11  });
12  afterAll ( Funktion(){
13    foo = 0 ;
14  });
15 });
```

Die Ausführliche Anleitung zur Jasmine-Framework kann man auf der Seite https://jasmine.github.io/pages/docs_home.html lesen.

5

Architektur

Dieses Kapitel handelt von der Umsetzung der Anforderungen und führt zur Darstellung von verschiedenen Diagrammen.

5.1 Architekturdiagramm

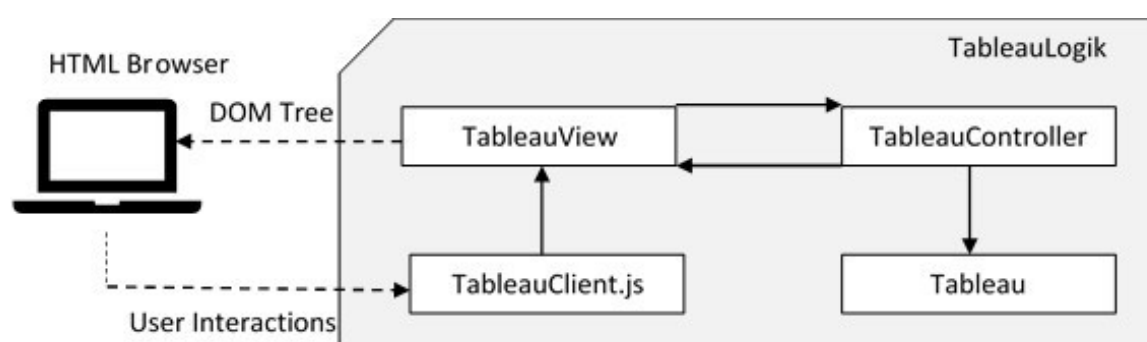


Abb. 5.1. Achitektur

Abbildung 5.1 zeigt die Architektur von der Tableau-Webanwendung im Überblick. Die Architektur folgt dem bewährten Model-View-Controller-Prinzip. Softwaretechnisch gliedert sich die Anwendungslogik so in mehrere Komponenten (Klassen) mit spezifischen funktionalen Verantwortlichkeiten. Die *TableauClient.js* kann Nutzerinteraktionen erkennen und über den View an den Controller weiterleiten. Der Controller unter entsprechenden Nutzerinteraktionen in Model umsetzen. Der Controller wird detailliert im Unterabschnitt 6.4.2 erläutert.

Die *TableauView* kapselt den DOM-Tree und bietet entsprechende Manipulationsmethoden für den Controller an, um sich verändernde Anwendungszustände im Browser zur Anzeige zu bringen. Der View wird im Unterabschnitt 6.4.1 erläutert.

Konzeptionell wird die Tableau-Webanwendung in einem Model abgebildet. Das Model ist komplexer und gliedert sich in mehrere logische Entities, die sich aus den Grundlagen des Kapitels 3 ableiten und im Unterabschnitt 6.3 erläutert werden.

5.2 Klassendiagramm

Um einen genaueren Überblick zu erhalten, welche Klassen wie mit einander kommunizieren, eignet sich ein Klassendiagramm. Nachfolgend ist zu erkennen, in welcher Kommunikation die Klassen zueinander stehen.

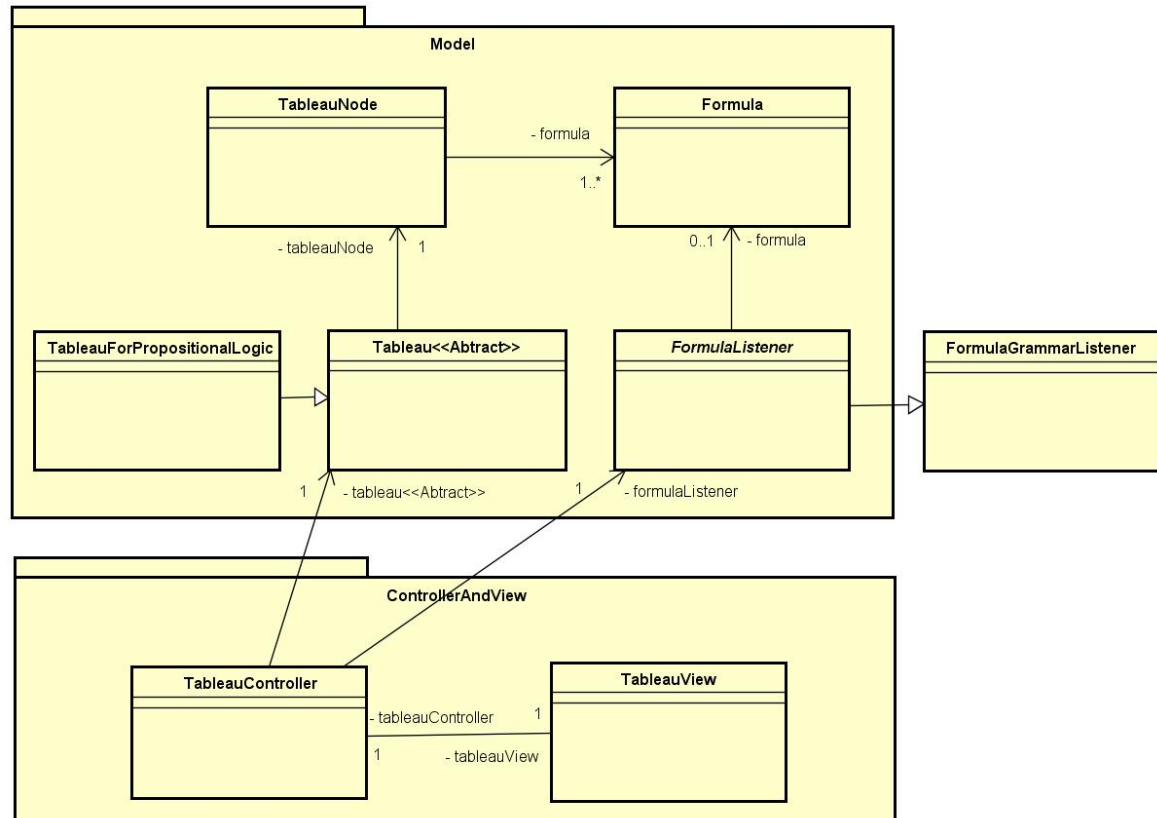


Abb. 5.2. Klassendiagramm

Tableau-Webanwendung umfasst verschiedene Bereiche:

- Analyse der Eingabeformel und Darstellung einer Formel als Baumstruktur (/LF10/): Klassen *FormulaListener* und *Formula*.
- Erstellung und Darstellung des Tableaus (/LF20/ und /LF30/): Klassen *TableauNode*, *Tableau* und *TableauForPropositionalLogic*.
- Verarbeitung der GUI-Interaktionen (/LF10/, /LF20/, /LF30/ und /LF40/): Klassen *TableauView* und *TableauController*.

Um die Möglichkeit zur Erweiterung der Software, gibt es eine abstrakte Klasse *Tableau*, welche die Tableaus erweitern. Die abstrakte Klasse definiert die Basismethoden um ein Tableau zu erstellen.

6

Implementierung

In diesem Kapitel geht es um die Implementierung der Webanwendung.

Wie im Abschnitt 4.1 erwähnt, kann man das Skript *require.js* verwenden um das Importieren von dutzenden Dateien manuell zu vermeiden, wie im folgenden Beispiel gezeigt:

```
1 <script src="src/Enums.js"></script>
2 <script src="src/Model/Formula.js"></script>
3 <script src="src/Model/TableauNode.js"></script>
4 <script src="src/Model/TableauForPropositionalLogic.js"></script>
5 ...
6 </body>
7 </html>
```

Diese Dateien müssen in der folgenden Reihenfolge geladen werden:

- *Enums.js* muss zuerst geladen werden, da alle anderen Dateien es benötigen.
- *Formula.js* wird von *TableauNode.js* verwendet und muss daher zuerst geladen werden.
- In ähnlicher Weise wird *TableauNode.js* von *TableauForPropositionalLogic.js* verwendet und muss daher als nächstes geladen werden.

Man kann leicht erkennen, dass das Laden von Dateien in der richtigen Reihenfolge wichtig ist, da sie voneinander abhängig sind. Dies mag zunächst kein Problem sein, aber wenn der Code kompliziert wird, wachsen die Dateien und es wird immer schwieriger, die *script*-Tags zu verwalten.

Daher wird *require.js* im Rahmen dieser Arbeit nicht nur für die ANTLR-Bibliothek sondern für alle andere JavaScript-Dateien verwendet. *require.js* implementiert die Funktion *require()* von *Node.js* im Browser. *require()* wird zum Laden von Dateien verwendet. Für normale JavaScript-Dateien bedeutet dies, dass sie ausgeführt werden, wenn sie zum ersten Mal geladen werden, und das war es. Der Code wird in seiner eigenen Schließung ausgeführt, so dass er den Rest des Codes nicht stört (z.B. identische Variablennamen sind kein Problem). Die einzige Möglichkeit, etwas an die Außenwelt

zurückzugeben, besteht darin, ein spezielles Objekt *exports*, das aufgerufen wird, zu ändern, wobei dies der Rückgabewert des *require()* Aufrufs ist. Eine JavaScript-Datei, die *exports* eine Funktion oder Variable an die Außenwelt zurückgibt, wird als “Modul” bezeichnet [Pix12]. Man kann *require()* in seinem Browser wie folgt verwenden:

```

1 <html>
2 <head>
3   <script type="text/javascript" src="lib/require.js"></script>
4 </head>
5 <body>
6   <script type="text/javascript">
7     var Formula = require("src/Model/Formula.js").Formula;
8     var formula = new Formula("1", null, null, FormulaTypeEnum.TRUE);
9     var type = formula.getFormulaType();
10  </script>
11 </body>
12 </html>

```

Der Code in *Formula.js* könnte so aussehen:

```

1 function Formula(label, left, right, formulaType) {
2   this.right = right;
3   this.left = left;
4   this.label = label;
5   this.formulaType = formulaType;
6 }
7 Formula.prototype.getFormulaType = function () {
8   return this.formulaType;
9 };
10
11 exports.Formula = Formula;

```

Die einzige Datei, die man normalerweise laden muss, ist *require.js*, die definiert *window.require()*. Danach kann man das Modul *Formula.js* wie in *Node.js* laden. Die Variable *Formula* enthält alle vom Modul exportierten Daten, so dass man sich diese als “Namespace” vorstellen kann. Das Aufrufen einer exportierten Funktion *getFormulaType()* in diesem Fall, funktioniert genauso wie das Aufrufen der Methode eines Objekts. Von hier werden alle exportierten Funktionen als “Methoden” bezeichnet.

Im Rahmen dieser Arbeit werden alle sogenannten “Namespaces” im Skript *Global.js* definiert. Weiterhin enthält das Skript auch alle globalen Variablen der Anwendung.

- *view*: globale Instanz der Klasse *TableauView*
- *InputError*: Boolean Variable, ist “true” wenn es eine Antlr-Fehlermeldung gibt.
- *ErrorMessage*: Ein Array um die Antlr-Fehlermeldungen zu speichern.
- *TableauChartData*: “Google Charts DataTable” um das Tableau zu visualisieren.
- *ArrOfTableauNodesByStepByStep*: Ein Array speichert die Tableau-Knoten für den Tableau-Chart in einem Schritt der “Schritt für Schritt Lösung” (wird in der Methode *drawTableauChartStepByStep()* verwendet).
- *ArrOfAllTableauNodes*: Ein Array speichert die übrige Tableau-Knoten, die noch nicht für den Tableau-Chart der “Schritt für Schritt Lösung” benutzt werden (wird in der Methode *drawTableauChartStepByStep()* verwendet).

- *TableauStepByStepData*: “Google Charts DataTable” um das Tableau für die “Schritt für Schritt Lösung” zu visualisieren.
- *stepNr*: Ordnungszahl des aktuellen Schrittes (wird für “Schritt für Schritt Lösung” verwendet).
- *chartNr*: Ordnungszahl des aktuellen Organigramm (wird für “Schritt für Schritt Lösung” verwendet).
- *nextStepNr*: Ordnungszahl des nächsten Schrittes (wird für “Schritt für Schritt Lösung” verwendet).
- *isTautology*: Ergebnis der Tautologie-Prüfung. Die Variable wird in der Enumeration *TautologyOrSatisfiableEnum* definiert (default: *TautologyOrSatisfiableEnum.NOTTESTED*, wenn die Tautologie nicht geprüft wird).
- *isSatisfiable*: Ergebnis der Erfüllbarkeit-Prüfung. Die Variable wird in der Enumeration *TautologyOrSatisfiableEnum* definiert (default: *TautologyOrSatisfiableEnum.NOTTESTED*, wenn die Erfüllbarkeit nicht geprüft wird).

```

1 const TautologyOrSatisfiableEnum = Object.freeze({
2   TRUE: "TRUE",
3   FALSE: "FALSE",
4   NOTTESTED: "NOTTESTED"
5 });
6 exports.TautologyOrSatisfiableEnum = TautologyOrSatisfiableEnum;

```

Listing 6.1. TautologyOrSatisfiableEnum

6.1 Struktur Dateisystem

Die Struktur der Dateien ist klar getrennt (Abbildung 6.1).

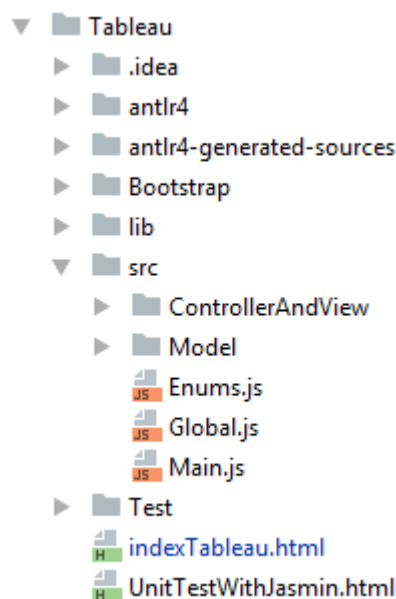


Abb. 6.1. Ordnerstruktur

In dem *“Tableau”* Ordner liegen alle Dateien, welche in die Seite eingebunden werden. Dazu zählt ANTLR- und Bootstrap-Framework, generierte Dateien von ANTLR, Bibliothek, JavaScript Dateien und Test. Neben dem Ordner liegen weitere Html Dateien. Dazu zählt die *“indexTableau.html”*, welche durch eingebundenes JavaScript interaktiv mit dem Anwender interagiert. Für das Testen des JavaScripts bietet es sich an, die HTML-Dateien *“UnitTestModel.html”* und *“UnitTestControllerView.html”* zu erstellen, die zusätzlich die Testfälle beinhaltet.

Der *“antlr4-generated-sources”* Ordner (Abbildung 6.2) erhält die ANTLR-Runtime, BAT-Datei um die Runtime auszuführen, Grammatik der aussagenlogischen Formeln und alle Dateien wie Token, Lexer, Parser, Listener und Visitor, welche die ANTLR-Runtime generiert hat.

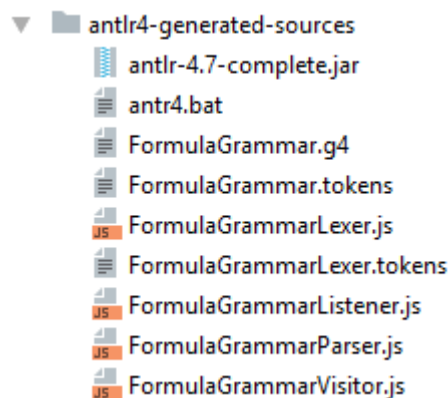


Abb. 6.2. antlr4GeneratedSources

Der *“lib”* Ordner (Abbildung 6.3) erhält die Jasmine-Bibliothek und *“require.js”*.

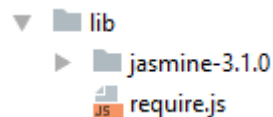


Abb. 6.3. lib

Für die Funktionalität der Webanwendung sorgen die JavaScript-Dateien aus dem *“src”* Ordner (Abbildung 6.4). Dazu zählt das Model, Controller und View sowie alle Enumerationen, globale Variable und *“Namespaces”*.

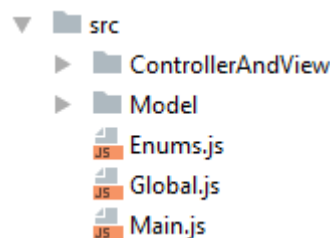


Abb. 6.4. src

Zur Qualitätssicherung gibt es Unit-Tests. Diese liegen unterhalb des *“Tests”* Ordners (Abbildung 6.5).

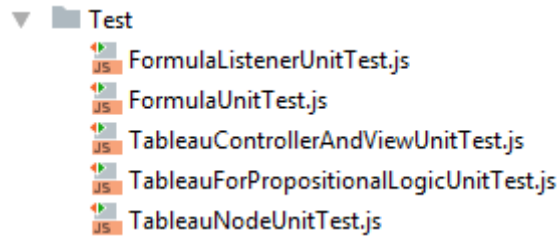


Abb. 6.5. test

6.2 FormulaGrammar.g4

Aus dem Unterabschnitt 3.1.6 kann eine Formel als Strings, die über eine kontextfreie formale Grammatik generiert wird, definiert werden. Das Alphabet der Formel, das in der Definition 3.3 definiert wird, wird wie folgt implementiert.

- Die Aussagenvariable kann beliebige große und kleine Buchstaben enthalten.

```
1 ATOM: [a-zA-Z];
```

- Basierend auf dem Unterabschnitt 3.1.5 werden die folgenden Notationen der Junktoren verwendet:

```
1 NEG : '!' ; // negation
2 IMP : '=>' ; // implication
3 EQU : '<=>' ; // equivalence
4 XOR : '^' ; // exclusive or
5 AND : '&' ; // conjunction
6 NAND : '!&' ; // nand
7 OR : '|' ; // disjunction
8 NOR : '!!' ; // nor
```

- Die Konstanten kann 0, 1, true oder false (in “case-insensitive”) sein.

```
1 TRUE: (T R U E | '1');
2 FALSE: (F A L S E | '0');
3 fragment T : [tT]; // match either an 't' or 'T'
4 fragment R : [rR];
5 fragment U : [uU];
6 fragment E : [eE];
7 ...
```

ANTLR löst Mehrdeutigkeiten zugunsten der zuerst genannten Alternative. Aus den Bindungsregeln in dem Unterabschnitt 3.1.3 hat die Regel *expr* z.B. die Konjunktionsalternative vor der Disjunktionsalternative. Die Produktionen von der Grammatik der aussagenlogischen Formeln wird wie folgt implementiert:

```
1 stat : expr EOF;
2 expr : NEG expr # Negation
3 | expr op=('&' | '!&') expr # AndNand
4 | expr op=('|' | '!!') expr # OrNor
5 | expr '=>' expr # Implication
6 | expr op('<=>' | '^') expr # EquXor
7 | ATOM # Atom
8 | TRUE # True
9 | FALSE # False
10 | '(' expr ')' # Parens
11 ;
```

Dazu sind “# Negation”, “# AndNand” usw. die Bezeichnungen von der Alternativen der Regel *expr*. Diese Bezeichnungen erscheinen am rechten Rand von Alternativen und beginnen mit dem # -Zeichen.

6.3 Model

Das Model repräsentiert die zugrunde liegende logische Struktur von Daten in der Tableau-Anwendung. Das Klassendiagramm des Models ist in Abbildung 6.6 gezeigt.

6.3.1 Formula

Aus der Definition 3.6 ist es möglich, dass jede Formel als ein Binärbaum dargestellt werden kann. Der Baum besteht aus Knoten (*Formula*). Jeder Knoten hat einen linken Teilbaum (*left*), einen rechten Teilbaum (*right*), die auch Knoten sind und einen Inhalt (*label*), welcher eine Aussagenvariable oder ein Junktore (Definition 3.3) ist [Kro16]. Die Junktoren werden in der Enumeration *FormulaOperatorEnum* definiert.

```

1  const FormulaOperatorEnum = Object.freeze({
2    NEG: "\u00AC",
3    AND: "\u2227",
4    OR: "\u2228",
5    NAND: "\u2191",
6    NOR: "\u2193",
7    IMP: "\u2192",
8    EQU: "\u2194",
9    XOR: "\u2295",
10 });
11 exports.FormulaOperatorEnum = FormulaOperatorEnum;
```

Listing 6.2. FormulaOperatorEnum

Weiterhin kann eine Formel ein Literal (Definition 3.11), α - oder β -Formel (Tabelle 3.2) sein. Diese Formeltypen werden in der Enumeration *FormulaTypeEnum* definiert und werden in dem Attribut *formulaType* der Klasse *Formula* gespeichert. Aus der Bemerkung 3.7 kann der Ausdruck als String von einer Formel durch eine In-order-Baumtraversierung erhalten werden. Um diesen Ausdruck korrekt abzubilden (z.B. $\neg q$ nicht $q\neg$), hat die Formel, die mit einer \neg (oder *FormulaOperatorEnum.NEG*) gekennzeichnet wird, immer nur ein rechtes Kind, d.h. *left* ist “Null”. Wenn eine Formel ein Blatt ist, sind beide (*left* und *right*) auch “Null”. Damit die Formel mittels Google Charts visualisiert werden kann (Unterabschnitt 4.3.1), muss jeder Knoten eine eindeutige Kennzeichnung (*id*) haben.

```

1  const FormulaTypeEnum = Object.freeze({
2    ALPHA: "ALPHA",
3    BETA: "BETA",
4    NEGATIVLITERAL: "NEGATIVLITERAL",
5    POSITIVLITERAL: "POSITIVLITERAL",
6    TRUE: "TRUE",
7    FALSE: "FALSE",
8  });
```

Listing 6.3. FormulaTypeEnum

Um die Tautologie einer Formel zu überprüfen, benötigt es die Negation der Formel. Daher bietet Klasse *Formula* die Methode *getNegativeFormula()*. Nach der Negation

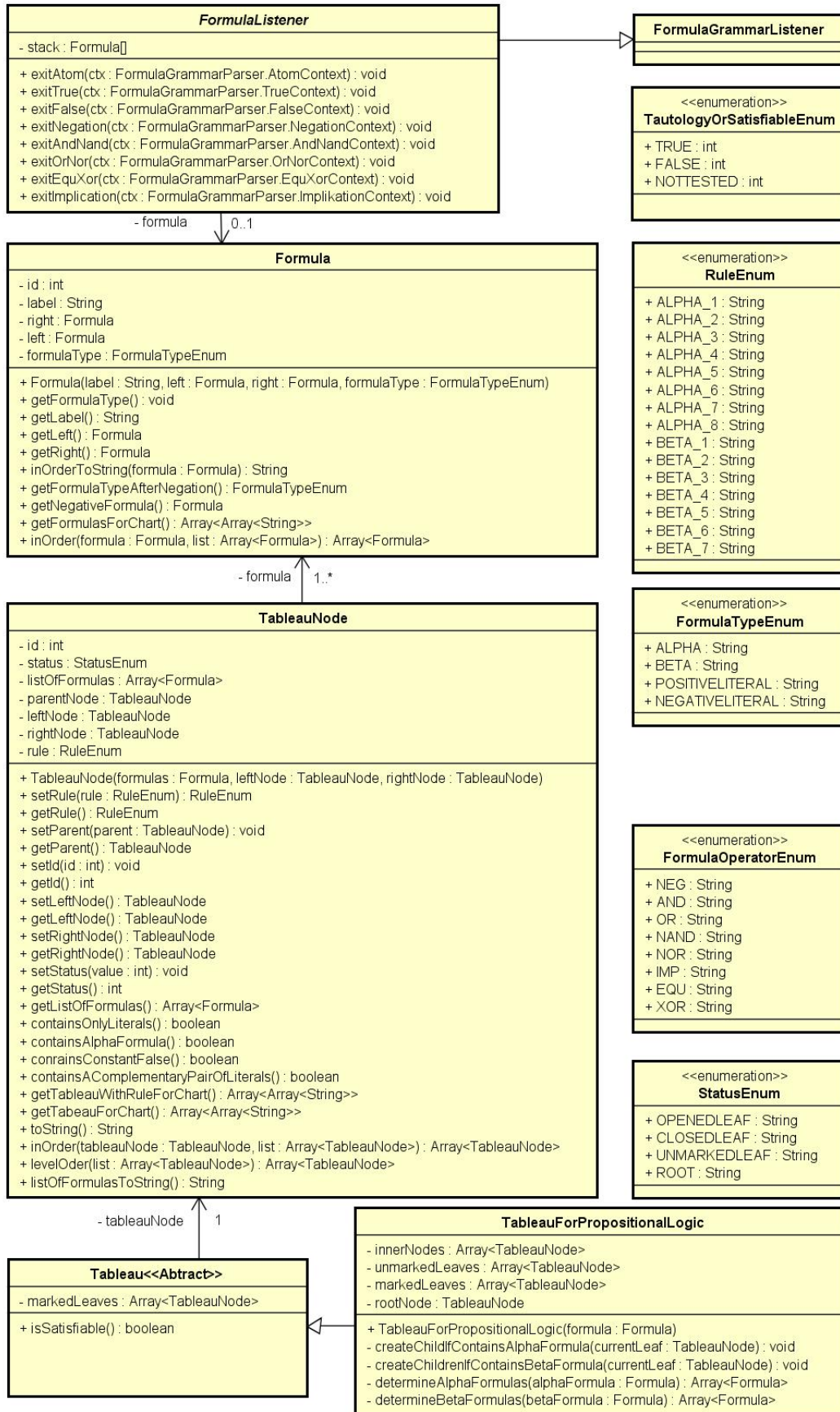


Abb. 6.6. Klassendiagramm(Model)

wird der Formeltyp geändert, deshalb wird die Methode *getFormulaTypeAfterNegation()* implementiert, um den neuen Formeltyp zu bestimmen.

```

1 Formula.prototype.getFormulaTypeAfterNegation = function () {
2   var formulaType;
3   if (this.getFormulaType() == FormulaTypeEnum.TRUE) {
4     formulaType = FormulaTypeEnum.FALSE;
5   } else if (this.getFormulaType() == FormulaTypeEnum.FALSE) {
6     formulaType = FormulaTypeEnum.TRUE;
7   } else if (this.getFormulaType() == FormulaTypeEnum.POSITIVLITERAL) {
8     formulaType = FormulaTypeEnum.NEGATIVLITERAL;
9   } else if (this.getLabel() == FormulaOperatorEnum.NEG) {
10    formulaType = FormulaTypeEnum.ALPHA;
11  } else if (this.getFormulaType() == FormulaTypeEnum.ALPHA) {
12    formulaType = FormulaTypeEnum.BETA;
13  } else {
14    formulaType = FormulaTypeEnum.ALPHA;
15  }
16  return formulaType;
17 };
18
19 Formula.prototype.getNegativeFormula = function () {
20   var negationFormulaType = this.getFormulaTypeAfterNegation();
21   var negationFormula = new Formula(FormulaOperatorEnum.NEG, null, this,
22     negationFormulaType);
23   return negationFormula;
24 };

```

Listing 6.4. *getNegativeFormula()* und *getFormulaTypeAfterNegation()* (Klasse *Formula*)

Außerdem bietet Klasse *Formula* die folgenden Methoden:

- *inOrder()*: Rekursive Methode, die eine Liste der Knoten durch In-Order-Baumtraversierung zurück gibt.
- *inOrderToString()*: Gibt Ausdruck als String von der Formel zurück.
- *getFormulasForChart()*: Erzeugt “Google Charts DataTable” um die Formel zu visualisieren. Die DataTable ist ein zweidimensionales Array. Jedes Element des Arrays beschreibt einen Knoten mit *id* des Knotens und der *id* des übergeordneten Knotens. Diese Ids werden nicht angezeigt. Im Diagramm wird nur das *label* des Knotens angezeigt. Alle Elemente des Arrays werden durch In-Order-Baumtraversierung sortiert.

```

1 Formula.prototype.getFormulasForChart = function () {
2   var ret = [];
3   var list = [];
4   var arr = [];
5   list = this.inOrder(this, list);
6   for (var i = 0; i < list.length; i++) {
7     list[i].id = i;
8   }
9   ret.push([v: this.id.toString(), f: this.label, ""]);
10  for (var i = 0; i < list.length; i++) {
11    if (list[i].left != null) {
12      arr = [v: list[i].left.id.toString(), f: list[i].left.label, list[i].id.
13        toString()];
14      ret.push(arr);
15    }
16    if (list[i].right != null) {
17      arr = [v: list[i].right.id.toString(), f: list[i].right.label, list[i].
18        id.toString()];
19      ret.push(arr);
20    }
21  }
22  return ret;
23 };

```

```

19     }
20     return ret;
21
22 };

```

Listing 6.5. getFormulasForChart()(Klasse Formula)

6.3.2 FormulaListener

Mit ANTLR kann man den Parse-tree mit einem benutzerdefinierten Listener oder einem benutzerdefinierten Visitor besuchen. Beide Implementierungen geben dasselbe Ergebnis aus. Die Visitor-Implementierung hat aber einen Vorteil, da die Visitor-Methoden einen Wert zurück geben und keine Werte in Feldern gespeichert werden müssen. Dagegen wird die Listener-Implementierung in dieser Arbeit gewählt, da diese im JavaScript bekannter ist und es viele Beispiele im Internet sowie auf der offiziellen Seite <https://github.com/antlr/antlr4/blob/master/doc/javascript-target.md> gibt.

FormulaListener definiert einen benutzerdefinierten Listener um die Syntaxbaum zu besuchen. Diese Klasse erweitert die Klasse *FormulaGrammarListener*, eine Klasse welche die ANTLR-Runtime automatisch generiert hat.

```

1  var FormulaListener = function () {
2      FormulaGrammarListener.call(this);
3  };
4
5  FormulaListener.prototype = Object.create(FormulaGrammarListener.prototype);

```

Listing 6.6. FormulaListener Konstruktor

FormulaGrammarListener definiert alle Methoden, die die Klasse *ParseTreeWalker* von der ANTLR-Runtime auslösen kann, wenn sie den Parse-tree durchläuft. Um die Eingabeformel zu analysieren muss man auf acht Ereignisse reagieren, indem man acht Methoden überschreibt: Wenn der Walker eine Aussagenvariable, einen Junktor oder eine Konstante verlässt. Hier sind die relevanten Methoden von der generierten Klassen *FormulaGrammarListener*:

```

1  function FormulaGrammarListener() {
2      antlr4.tree.ParseTreeListener.call(this);
3      return this;
4  }
5
6  FormulaGrammarListener.prototype = Object.create(antlr4.tree.ParseTreeListener.prototype);
7
8  FormulaGrammarListener.prototype.constructor = FormulaGrammarListener;
9  // Exit a parse tree produced by FormulaGrammarParser#Atom.
10 FormulaGrammarListener.prototype.exitAtom = function(ctx) {};
11
12 // Exit a parse tree produced by FormulaGrammarParser#True.
13 FormulaGrammarListener.prototype.exitTrue = function(ctx) {};
14
15 // Exit a parse tree produced by FormulaGrammarParser#False.
16 FormulaGrammarListener.prototype.exitFalse = function(ctx) {};
17
18 // Enter a parse tree produced by FormulaGrammarParser#Negation.
19 FormulaGrammarListener.prototype.enterNegation = function(ctx) {};
20
21 // Exit a parse tree produced by FormulaGrammarParser#AndNand.
22 FormulaGrammarListener.prototype.exitAndNand = function(ctx) {};
23
24 // Exit a parse tree produced by FormulaGrammarParser#OrNor.

```

```

25 FormulaGrammarListener.prototype.exitOrNor = function(ctx) {};
26
27 // Exit a parse tree produced by FormulaGrammarParser#EquXor.
28 FormulaGrammarListener.prototype.exitEquXor = function(ctx) {};
29
30 // Exit a parse tree produced by FormulaGrammarParser#Implication.
31 FormulaGrammarListener.prototype.exitImplication = function(ctx) {};

```

Listing 6.7. FormulaGrammarListener

Da Listener-Methoden keinen Wert zurück geben, muss *FormulaListener* ein Array *stack* um die besuchende Formeln zu speichern, haben. Weiterhin hat *FormulaListener* folgende überschriebenen Methoden. Jede Methode hat ein Argument *ctx*. Das Argument *ctx* ist eine Instanz eines spezifischen Klassenkontextes für den Knoten, den Walker verlässt.

- *exitAtom()*: Diese Methode fügt eine neue Aussagenvariable im *stack* hinzu wenn der Walker eine *Atom*-Regel verlässt.

```

1 FormulaListener.prototype.exitAtom = function (ctx) {
2     var label = ctx.ATOM().getText();
3     var formula = new Formula(label, null, null, FormulaTypeEnum.POSITIVLITERAL)
4     ;
5     this.stack.push(formula);
6 }

```

Listing 6.8. exitAtom() (Klasse FormulaListener)

- *exitTrue()*: Diese Methode fügt eine neue Konstante “true” im *stack* hinzu wenn der Walker eine *True*-Regel verlässt.

```

1 FormulaListener.prototype.exitTrue = function (ctx) {
2     var label = ctx.TRUE().getText().toLowerCase();
3     var formula = new Formula(label, null, null, FormulaTypeEnum.TRUE);
4     this.stack.push(formula);
5 }

```

Listing 6.9. exitTrue() (Klasse FormulaListener)

- *exitFalse()*: Analog wie *exitTrue()*.
- *exitNegation()*: Diese Methode fügt eine negierte Formel im *stack* hinzu wenn der Walker eine *Negation*-Regel verlässt.

```

1 FormulaListener.prototype.exitNegation = function (ctx) {
2     var label = FormulaOperatorEnum.NEG;
3     var formula;
4     var right = this.stack.pop();
5     if (right.getFormulaType() == FormulaTypeEnum.TRUE) {
6         formula = new Formula(label, null, right, FormulaTypeEnum.FALSE);
7     } else if (right.getFormulaType() == FormulaTypeEnum.FALSE) {
8         formula = new Formula(label, null, right, FormulaTypeEnum.TRUE);
9     } else if (right.getFormulaType() == FormulaTypeEnum.POSITIVLITERAL) {
10        formula = new Formula(label, null, right, FormulaTypeEnum.NEGATIVLITERAL);
11    } else if (right.getLabel() == FormulaOperatorEnum.NEG) {
12        formula = new Formula(label, null, right, FormulaTypeEnum.ALPHA);
13    } else if (right.getFormulaType() == FormulaTypeEnum.ALPHA) {
14        formula = new Formula(label, null, right, FormulaTypeEnum.BETA);
15    } else {
16        formula = new Formula(label, null, right, FormulaTypeEnum.ALPHA);
17    }
18    this.stack.push(formula);
19 }

```

Listing 6.10. exitNegation() (Klasse FormulaListener)

- *exitAndNand()*: Diese Methode fügt eine neue Formel im *stack* hinzu wenn der Walker eine *AndNand*-Regel verlässt.

```

1 FormulaListener.prototype.exitAndNand = function (ctx) {
2   var formula;
3   var right = this.stack.pop();
4   var left = this.stack.pop();
5   if (ctx.op.type === FormulaGrammarParser.AND) {
6     ;
7     formula = new Formula(FormulaOperatorEnum.AND, left, right,
8       FormulaTypeEnum.ALPHA);
9   } else {
10    formula = new Formula(FormulaOperatorEnum.NAND, left, right,
11      FormulaTypeEnum.BETA);
12  }
13  this.stack.push(formula);
14 };

```

Listing 6.11. *exitAndNand()* (Klasse *FormulaListener*)

- *exitOrNor()*, *exitEquXor()* und *exitImplication()*: Analog wie *exitAndNand()*.

6.3.3 TableauNode

Die Klasse *TableauNode* ist analog wie *Formula* als ein Binärbaum dargestellt (Definition 3.13). Ein Tableau-Knoten hat einen übergeordneten Knoten (*parentNode*), einen linken Teilbaum (*leftNode*), einen rechten Teilbaum (*rightNode*), eine eindeutige Kennzeichnung (*id*) um die Darstellung mittels Google Charts visualisieren zu können und eine Regel-Bezeichnung (*rule*), die in der Enumeration *RuleEnum* definiert.

```

1 const RuleEnum = Object.freeze({
2   ALPHA_1: "a1",
3   ALPHA_2: "a2",
4   ALPHA_3: "a3",
5   ALPHA_4: "a4",
6   ALPHA_5: "a5",
7   ALPHA_6: "a6",
8   ALPHA_7: "a7",
9   ALPHA_8: "a8",
10  BETA_1: "b1",
11  BETA_2: "b2",
12  BETA_3: "b3",
13  BETA_4: "b4",
14  BETA_5: "b5",
15  BETA_6: "b6",
16  BETA_7: "b7",
17 });
18 exports.RuleEnum = RuleEnum;

```

Listing 6.12. *RuleEnum*

Jeder *TableauNode* wird mit einem Array von Formeln (*listOfFormulas*) beschriftet. Falls *listOfFormulas* nur die Eingabeformel enthält, dann hat der Tableau-Knoten keinen *parentNode*. In jedem Fall hat er einen, zwei oder keinen untergeordneten Knoten. Wenn er nur einen untergeordneten Knoten hat, ist *leftNode* "Null". Außerdem erhält der Tableau-Knoten einen Zustand (*status*), die in der Enumeration *StatusEnum* definiert. Da Tableau ein Baum ist, ist ein *TableauNode* eine Wurzel (*StatusEnum.ROOT*) oder ein Blatt. Wenn es ein Blatt ist, dann kann es ein unmarkiertes (*StatusEnum.UNMARKEDLEAF*), offenes (*StatusEnum.OPENEDLEAF*) oder geschlossenes Blatt (*StatusEnum.CLOSEDLEAF*) sein (Algorithmus 3.15).

```

1 const StatusEnum = Object.freeze({
2   OPENEDLEAF: "OPENEDLEAF",
3   CLOSEDLEAF: "CLOSEDLEAF",
4   UNMARKEDLEAF: "UNMARKEDLEAF",
5   ROOT: "ROOT"
6 });
7 exports.StatusEnum = StatusEnum;

```

Listing 6.13. StatusEnum

Ein *TableauNode* kann:

- mittels Methode *containsOnlyLiterals()* überprüfen, ob er nur die Literalen erhält.

```

1 TableauNode.prototype.containsOnlyLiterals = function () {
2   for (var i = 0; i < this.listOfFormulas.length; i++) {
3     if (this.listOfFormulas[i].getFormulaType() == FormulaTypeEnum.ALPHA ||
4         this.listOfFormulas[i].getFormulaType() == FormulaTypeEnum.BETA) {
5       return false;
6     }
7   }
8   return true;
9 };

```

Listing 6.14. containsOnlyLiterals() (Klasse TableauNode)

- mittels Methode *containsAlphaFormula()* überprüfen, ob er mindestens eine α -Formel erhält.

```

1 TableauNode.prototype.containsAlphaFormula = function () {
2   for (var i = 0; i < this.listOfFormulas.length; i++) {
3     if (this.listOfFormulas[i].getFormulaType() == FormulaTypeEnum.ALPHA) {
4       return true;
5     }
6   }
7   return false;
8 };

```

Listing 6.15. containsAlphaFormula() (Klasse TableauNode)

- mittels Methode *containsConstantFalse()*überprüfen, ob er mindestens eine Konstante 0 der “false” (*FormulaTypeEnum.FALSE*) erhält.

```

1 TableauNode.prototype.containsConstantFalse = function () {
2   for (var i = 0; i < this.listOfFormulas.length; i++) {
3     if (this.listOfFormulas[i].getFormulaType() == FormulaTypeEnum.FALSE) {
4       return true;
5     }
6   }
7   return false;
8 };

```

Listing 6.16. containsConstantFalse() (Klasse TableauNode)

- mittels Methode *containsAComplementaryPairOfLiterals()*überprüfen, ob er mindestens ein komplementäres Paar erhält.

```

1 TableauNode.prototype.containsAComplementaryPairOfLiterals = function () {
2   var positiveLiterals = [];
3   var negativeLiterals = [];
4   for (var i = 0; i < this.listOfFormulas.length; i++) {
5     if (this.listOfFormulas[i].getFormulaType() == FormulaTypeEnum.
6         POSITIVLITERAL) {
7       if (negativeLiterals.length == 0) {
8         positiveLiterals.push(this.listOfFormulas[i]);
9       }
10    }
11    else if (this.listOfFormulas[i].getFormulaType() == FormulaTypeEnum.
12        NEGATIVLITERAL) {
13      if (positiveLiterals.length == 0) {
14        negativeLiterals.push(this.listOfFormulas[i]);
15      }
16    }
17  }
18  return (positiveLiterals.length > 0 & negativeLiterals.length > 0);
19 };

```



```

10         if (negativeLiterals[j].label == this.listOfFormulas[i].getLabel
11             ()) {
12             return true;
13         } else {
14             positiveLiterals.push(this.listOfFormulas[i]);
15         }
16     } else if (this.listOfFormulas[i].getFormulaType() == FormulaTypeEnum.
NEGATIVLITERAL) {
17         if (positiveLiterals.length == 0) {
18             negativeLiterals.push(this.listOfFormulas[i].getRight());
19         }
20         for (var h = 0; h < positiveLiterals.length; h++) {
21             if (positiveLiterals[h].label == this.listOfFormulas[i].getRight
22                 ().getLabel()) {
23                 return true;
24             } else {
25                 negativeLiterals.push(this.listOfFormulas[i].getRight());
26             }
27         }
28     }
29 }
30 return false;
31 };

```

Listing 6.17. containsConstantFalse() (Klasse TableauNode)

Weiterhin bietet Klasse *TableauNode* die folgenden Methoden:

- *inOrder()*: Rekursive Methode, die eine Liste der Tableau-Knoten durch In-Order-Baumtraversierung zurück gibt.
- *levelOrder()*: Gibt eine Liste der Tableau-Knoten durch Breitensuche zurück.
- *toString()*: Führt eine In-Order-Baumtraversierung des Tableau-Knotens durch und gibt das Ergebnis als String zurück.
- *listOfFormulasToString()*: Gibt alle Formeln, die der Tableau-Knoten erhält, zurück.
- *getTableauForChart()*: Erzeugt “Google Charts DataTable” um das Tableau zu visualisieren. Die DataTable ist ein zweidimensionales Array. Jedes Element des Arrays beschreibt einen Knoten mit *id* des Knotens und der *id* des übergeordneten Knotens. Diese Ids werden nicht angezeigt. Im Diagramm werden die Formeln, die der Tableau-Knoten erhält, \times oder \odot angezeigt. Alle Elemente des Arrays werden durch In-Order-Baumtraversierung sortiert.
- *getTableauWithRuleForChart()*: Analog wie *getTableauForChart()*, gibt es aber noch Regel-Bezeichnungen (*rule*) anzuzeigen. Alle Elemente des Arrays werden durch Breitensuche sortiert, um diese später für die “Schritt für Schritt Lösung” zu verwenden.

6.3.4 Tableau

Tableau ist eine abstrakte Klasse, die die Basismethode *isSatisfiable()* bietet. Diese Methode prüft ob das Array des markierten Blattes (*markedLeaves*) ein offenes Blatt enthält. Wenn ja, ist die Eingabeformel erfüllbar, sonst ist sie unerfüllbar.

```

1 Tableau.prototype.isSatisfiable = function(){
2   for(var i = 0; i < this.markedLeaves.length; i++){
3     if(this.markedLeaves[i].getStatus() == StatusEnum.OPENEDLEAF){
4       return true;
5     }
6   }
7   return false;
8 };
9
10 exports.Tableau = Tableau;

```

Listing 6.18. isSatisfiable() (Klasse Tableau)

6.3.5 TableauForPropositionalLogic

TableauForPropositionalLogic erweitert die Klasse *Tableau*. Konstruktor der *TableauForPropositionalLogic* Klasse hat ein Argument *formula*, eine Instanz der Klasse *Formula*, die die Eingabeformel implementiert. Dieser Konstruktor setzt die Konstruktion eines semantischen Tableaus für Aussagenlogik (Algorithmus 3.15) um. Dazu speichert er die Wurzel des Tableau im Attribut *rootNode*, ein Array des unmarkierten Blattes im Attribut *unmarkedLeaves* und ein Array des markierten Blattes im Attribut *markedLeaves*.

```

1 function TableauForPropositionalLogic(formula) {
2   this.innerNode = [];
3   this.unmarkedLeaves = [];
4   this.markedLeaves = [];
5   var formulasArr = [formula];
6   var rootNode = new TableauNode(formulasArr, null, null);
7   idNr++;
8   rootNode.setId(idNr);
9   this.unmarkedLeaves.push(rootNode);
10  while (this.unmarkedLeaves.length > 0) {
11    var currentLeaf = this.unmarkedLeaves.pop();
12    if (currentLeaf.containsOnlyLiterals()) {
13      if (currentLeaf.containsAComplementaryPairOfLiterals() || currentLeaf.
containsConstantFalse()) {
14        currentLeaf.setStatus(StatusEnum.CLOSEDLEAF);
15      } else {
16        currentLeaf.setStatus(StatusEnum.OPENEDLEAF);
17      }
18      this.markedLeaves.push(currentLeaf);
19    } else if (currentLeaf.containsAlphaFormula()) {
20      this.createChildIfContainsAlphaFormula(currentLeaf);
21    } else {
22      this.createChildrenIfContainsBetaFormula(currentLeaf);
23    }
24  }
25  if (this.innerNode.length == 0) {
26    this.rootNode = this.markedLeaves[0];
27  } else {
28    this.rootNode = this.innerNode[0];
29  }
30 }
31
32
33 TableauForPropositionalLogic.prototype = Object.create(Tableau.prototype);

```

Listing 6.19. TableauForPropositionalLogic Konstruktor

Um diese Tableau-Implementierung übersichtlicher zu gestalten und Fehler zu vermeiden, hat *TableauForPropositionalLogic* die folgenden Methoden:

- *determineAlphaFormulas()*: implementiert die Zeile 13 des Algorithmus 3.15. Dazu erhält sie eine α -Formel als Argument. Durch die Berücksichtigung des Hauptoperator sowie die Negation (falls es eine gibt) der Argument-Formel, bestimmt *determineAlphaFormulas()* die entsprechende Regel nach Tabelle 3.2 und ermittelt Formeln α_1 und α_2 .
- *determineBetaFormulas()*: Analog *determineAlphaFormulas()*, implementiert sie die Zeile 19 des Algorithmus 3.15. Diese Methode gibt ein Array, das die Formeln β_1 , β_2 und die verwendete Regel enthält, in Reihenfolge zurück.
- *createChildIfContainsAlphaFormula()*: Erhält einen Tableau-Knoten *currentLeaf* und implementiert die Zeile 13 bis Zeile 16 des Algorithmus 3.15. Dazu wählt sie die erste α -Formel des *currentLeaf*. Mit dieser Formel kann sie mittels Methode *determineAlphaFormulas()* die Informationen der untergeordneten Knoten (wie α_1 , α_2 und verwendete Regel), den sie danach erzeugen muss, bekommen. Der untergeordnete Knoten wird mit einem Array der Formeln (*listOfFormula*) beschriftet. Diese Array enthält die Formeln aus *currentLeaf* nach dem Ersetzen der ausgewählten α -Formel durch Formeln α_1 , α_2 .
- *createChildrenIfContainsBetaFormula()*: Analog wie *createChildIfContainsAlphaFormula()*, implementiert sie die Zeile 19 bis Zeile 24 des Algorithmus 3.15. Dazu erzeugt sie mit Hilfe der Methode *determineBetaFormulas()* zwei untergeordnete Knoten von *currentLeaf*. Der linke untergeordnete Knoten (*leftNode*) wird mit einem Array, das die Formeln aus *currentLeaf* nach dem Ersetzen der ausgewählten β -Formel durch β_1 enthält, beschriftet. Der rechte untergeordnete Knoten (*rightNode*) wird mit einem Array, das die Formeln aus *currentLeaf* nach dem Ersetzen der ausgewählten β -Formel durch β_2 enthält, beschriftet.

6.4 View und Controller

6.4.1 View

Der View dient der Darstellung der Tableau-Anwendung für den Anwender. Er besteht im Kern aus einem HTML-Dokument (Unterabschnitt 6.4.1.1) und einer clientseitigen Logik, die den DOM-Tree des HTML-Dokuments manipuliert. Die clientseitige Logik hat zwei Bestandteile, also *TableauClient.js* (Unterabschnitt 6.4.1.2) und Klasse *TableauView* (Unterabschnitt 6.4.1.3).

6.4.1.1 HTML-Dokument

Das folgendes HTML-Dokument (Listing 6.20 und 6.21) wird genutzt, um die Anwendung einzublenden. Abbildung 6.7 zeigt dabei einen Screenshot der Anwendung in der Desktop- und Mobile-Version.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">

```

```

5 <meta http-equiv="X-UA-Compatible" content="IE=edge">
6 <meta name="viewport" content="width=device-width, initial-scale=1">
7 <!-- The above 3 meta tags *must* come first in the head; any other head content
   must come *after* these tags -->
8 <title>Tableau Demo</title>
9 <!-- Bootstrap -->
10 <link href="Bootstrap/css/bootstrap.min.css" rel="stylesheet">
11 <link href="Bootstrap/css/bootstrap-theme.min.css" rel="stylesheet">
12 <link href="Bootstrap/style.css" rel="stylesheet">
13 <script src="lib/require.js"></script>
14 <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></
   script>
15 <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js">
   </script>
16 <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></
   script>
17 </head>

```

Listing 6.20. HTML-head

Diese *head*-Tags enthalten den Titel für benötigte Dokumente, Skripte und Metainformationen. In den *link*-Tags wird Bootstrap-Framework und das benutzerdefinierte Stylesheet *style.css* verknüpft. In den *script*-Tags werden die clientseitigen Skripts definiert. Hierbei wird *require.js* verwendet um JS-Dateien zu importieren. *jquery.min.js* und *bootstrap.min.js* werden verwendet um jQuery-Bibliothek und Bootstrap-JavaScript-Plugins einzubinden und *loader.js* wird verwendet um Google-Chart Loader zu laden.

```

1 <body>
2 <div id="header" ...>
3 <div id="main">
4   <div id="mainContainer" class="container">
5     <div class="row">
6       <div class="col-md-9 content">
7         <div class="row" id="formula-input">
8           <div class="col-md-12">
9             <form action="" class="form-horizontal" onSubmit="return false;">
10              <div class="form-group">
11                <div class="col-md-12">
12                  <input id="input" class="form-control input-lg" ...>
13                  [...]
14                <div class="form-group">
15                  <div class="col-md-12">
16                    <button class="btn btn-warning btn-lg" type="button"
17                      onclick="pressSatisfiableButton()">
18                      Erfüllbarkeit
19                    </button>
20                    <button class="btn btn-warning btn-lg" type="button"
21                      onclick="pressTautologyButton()">
22                      Tautologie
23                  [...]
24                <div id="panel-input" class="col-md-12">
25                  <div class="panel panel-info" data-toggle="collapse" data-target="#panel-
   -input-body">
26                    <div id="panel-input-header" class="panel-heading" ...>
27                    <div id="panel-input-body" class="collapse in panel-body">
28                      <div id="formulaInput" ...>
29                      <div id="errorInput" class="text-danger"></div>
30                    [...]
31                  <div id="panel-result" class="col-md-12">
32                    <div class="panel panel-info">
33                      <div id="panel-result-header" class="panel-heading" ...>
34                      <div id="result" class="panel-body collapse in">
35                        <div class="col-md-9" style="padding-left: 0px">
36                          <div id="satisfiableDiv" ...>
37                          <div id="unsatisfiableDiv" ...>
38                          <div id="tautologyDiv" ...>
39                        </div>
40                      <div class="col-md-3" style="padding-right:0px; margin-right:0px">

```

```

41 <button class="btn btn-warning pull-right" type="button" onclick="
    stepByStepSolution()" ...>
42     Schritt für Schritt Lösung
43 </button>
44 <div class="modal fade" id="solutionModal" role="dialog" ...>
45     [...]
46 <div id="panel-formula" class="col-md-12">
47     <div class="panel panel-info">
48         <div id="panel-formula-header" class="panel-heading" ...>
49         <div id="Formula_chart" class="collapse in panel-body scroll">
50             <div id="Formula_chart_div" class="chart"></div>
51         [...]
52 <div id="panel-tableau" class="col-md-12">
53     <div class="panel panel-info">
54         <div id="panel-tableau-header" class="panel-heading" ...>
55         <div id="Tableau_chart" class="collapse in panel-body scroll">
56             <div id="Tableau_chart_div" class="chart"></div>
57         [...]
58 <div class="col-md-3 sidebar">
59 </div>
60 </div>
61 </div>
62 <div id="footer">
63 <script>...</script>
64 <script src="src/Global.js"></script>
65 <script src="src/TableauClient.js"></script>
66 </body>

```

Listing 6.21. HLML-body

Wie in Unterabschnitt 4.2.4 erwähnt, ist die Benutzeroberfläche der Tableau- Anwendung als eine Responsive-Website mit “Laptop-First” Strategie (Bildschirmgröße ≥ 768 px) implementiert. Daher wird das Layout der Seite basierend auf der Klasse *.col-md-* von der Bootstrap Grid-System angeordnet und angepasst.

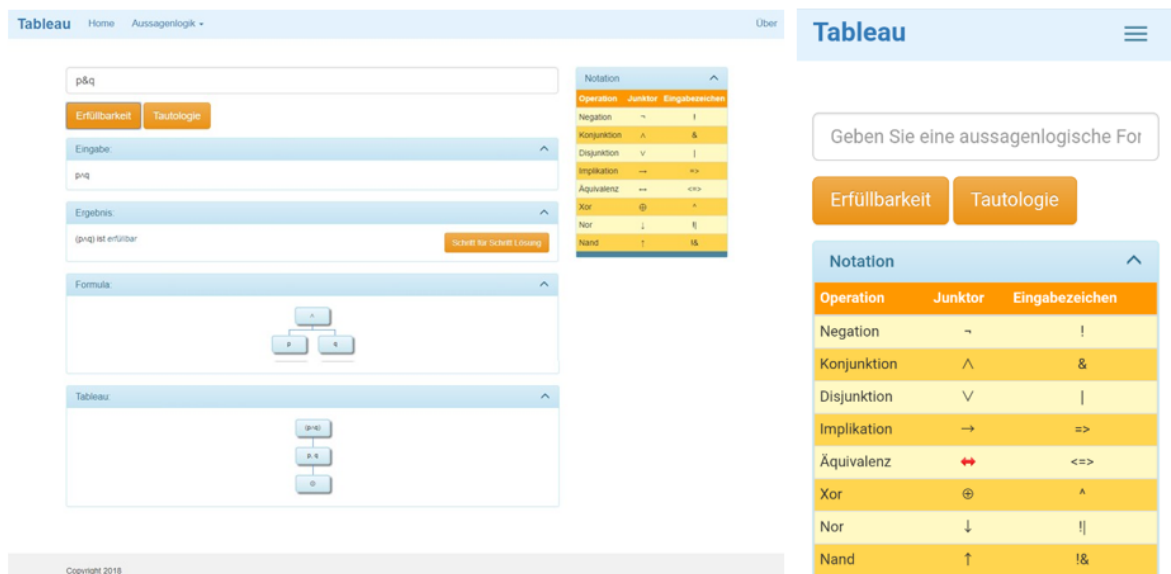


Abb. 6.7. Screenshots der Tableau-Anwendung für Desktop-(links) und Mobile-Version(rechts)

6.4.1.2 TableauClient.js als Schnittstelle zum HTML-Dokument

Der View wird im Browser initial durch obiges HTML-Dokument erzeugt. Im Verlaufe der Anwendung wird der DOM-Tree dieses HTML-Dokuments durch die Klasse

TableauView manipuliert, um den Anwendungszustand darzustellen. Die Klasse *TableauView* wird dabei durch das Skript *TableauClient.js*, das die Nutzerinteraktionen ermöglicht, geladen. In dem Skript *TableauClient.js* werden die folgenden Funktionen implementiert. Hier gibt es eine Beachtung, dass diese Funktionen wie normale JavaScript-Funktionen implementiert werden und nicht vom einem Objekt-Namespaces z.B. *MyObj.theMethod()* aufgerufen werden.

- *pressTautologyButton()*: Diese Funktion wird ausgeführt, wenn der Anwender auf den Button “Tautologie” drückt. Sie initialisiert eine Instanz der Klasse *TableauView* und ruft die Methode *checkTautology()* der Instanz auf.
- *pressSatisfiableButton()*: Diese Funktion wird ausgeführt, wenn der Anwender auf den Button “Erfüllbarkeit” drückt. Sie initialisiert eine Instanz der Klasse *TableauView* und ruft die Methode *checkSatisfiable()* der Instanz auf.
- *stepByStepSolution()*: Diese Funktion wird ausgeführt, wenn der Anwender auf den Button “Schritt für Schritt Lösung” drückt. Über dem Aufruf der Methode *stepByStepSolutionOutput()* von der Klasse *TableauView* leitet sie diese Benutzerinteraktion an eine Instanz der Klasse *TableauView* weiter.
- *pressNextStepButton()*: Diese Funktion wird ausgeführt, wenn der Anwender auf den Button “Nächste Schritt” mit der Id *nextStepButton* des DOM-Trees drückt. Über dem Aufruf der Methode *nextSolutionStepOutput()* von der Klasse *TableauView* leitet sie diese Benutzerinteraktion an eine Instanz der Klasse *TableauView* weiter.

Außerdem enthält *TableauClient.js* die folgenden Funktionen. Die Funktionen werden von Google Charts unterstützt um die Organigramme auf der Webseite zeichnen zu lassen.

- *drawFormulaChart()*: Visualisiert die Darstellung der Formel mittels Google Charts. Diese Darstellung wird in das DIV-Element mit der Id *Formula_chart_div* des DOM-Trees eingeblendet.
- *drawTableauChart()*: Visualisiert die Darstellung des Tableau ohne Regel-Bezeichnungen mittels Google Charts. Diese Darstellung wird in das DIV-Element mit der Id *Tableau_chart_div* des DOM-Trees eingeblendet.
- *drawTableauChartStepByStep()*: Visualisiert teilweise die Darstellung des Tableau mittels Google Charts.
- *drawTableauChartWithRule()*: Visualisiert die Darstellung des Tableau mit Regel-Bezeichnungen mittels Google Charts. Diese Darstellung wird in das DIV-Element mit der Id *stepByStepDiagram* des DOM-Trees eingeblendet.

```

1 function drawTableauChart() {
2     TableauChartData = new google.visualization.DataTable();
3     var arrOfTableauNodes = view.controller.rootTableau.getTableauForChart();
4     // For each orgchart box, provide the label, and the parentNode
5     TableauChartData.addColumn('string', 'Label');
6     TableauChartData.addColumn('string', 'Parent');
7     TableauChartData.addRows(arrOfTableauNodes);

```



```

8  // Create the chart.
9  var chart = new google.visualization.OrgChart(document.getElementById('
Tableau_chart_div'));
10 // Draw the chart, setting the allowHtml option to true for the tooltips.
11 chart.draw(TableauChartData, {allowHtml: true});
12 }

```

Listing 6.22. drawTableauChart() (TableauClient.js)

6.4.1.3 TableauView

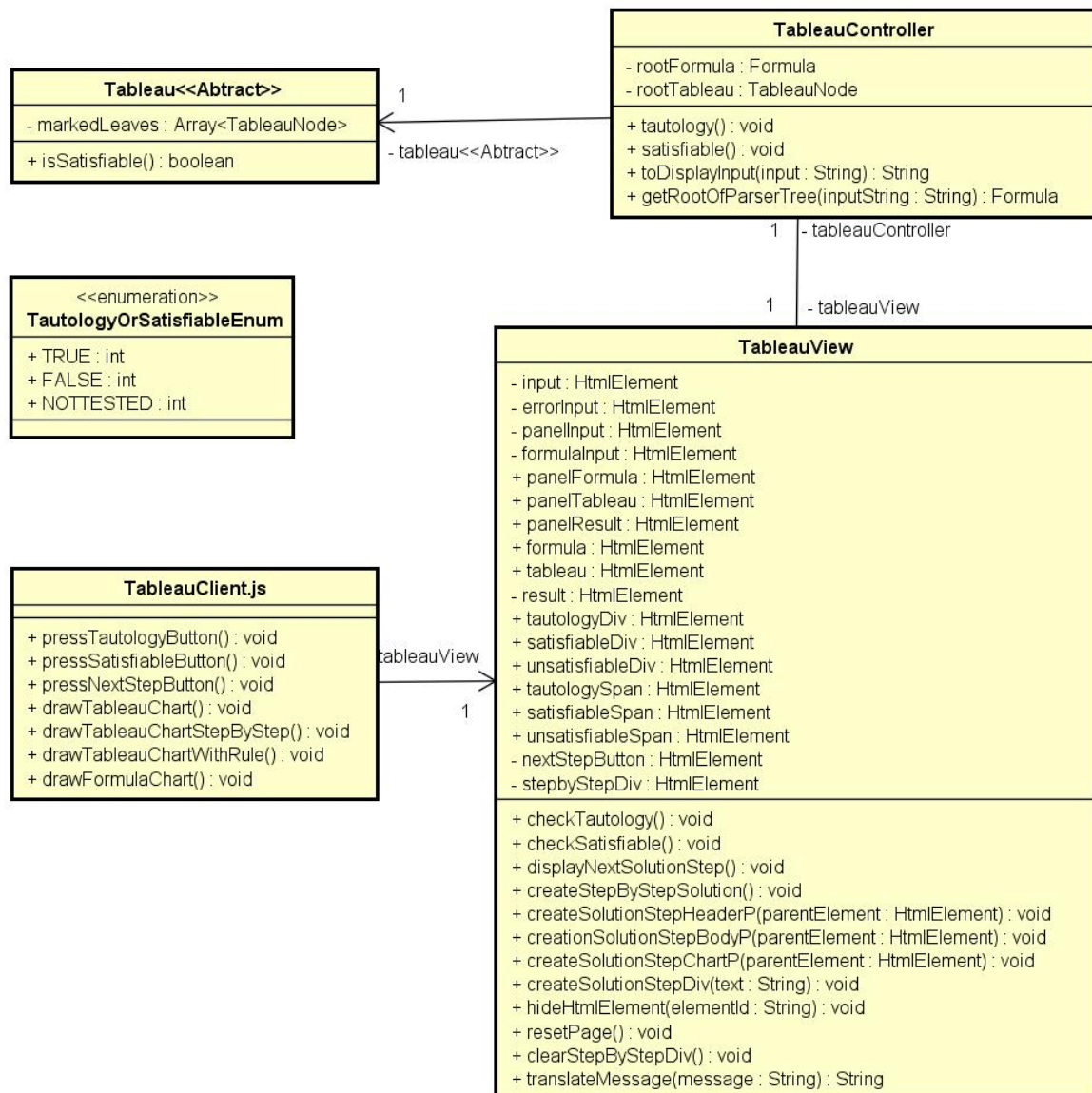


Abb. 6.8. Klassendiagramm (View und Controller)

Folgende Elemente haben dabei eine besondere Bedeutung und können dabei über entsprechende Attribute der Klasse *TableauView* (Abbildung 6.8) angesprochen werden.

- *input*: Das Element mit dem Identifier *input* dient dazu die Eingabe zu erhalten.

- *errorInput*: Das Element mit dem Identifier *errorInput* dient dazu ein Warnungstext bei fehlerhaften Eingaben einzublenden.
- *panelInput*: Das Element mit dem Identifier *panel-input* wird genutzt, um die Tafel, die die Eingabeformel enthält, einzublenden.
- *formulaInput*: Das Element mit dem Identifier *formulaInput* dient dazu die Eingabeformel einzublenden.
- *panelFormula*, *panelTableau* und *panelResult* : Die Elemente mit dem Identifier *panel-formula*, *panel-tableau* und *panel-result* werden genutzt, um die Tafeln, die die Darstellung der Formel, die Darstellung des Tableau und das Prüfungsergebnis (z.B. dass die Eingabeformel erfüllbar oder Tautologie ist) enthält, einzublenden.
- *formula*, *tableau* und *result* : Die Elemente mit dem Identifier *formula*, *tableau* und *result* werden genutzt, um die Panel-Bodys der Panel *panel-formula*, *panel-tableau* und *panel-result* einzublenden.
- *tautologyDiv*, *satisfiableDiv* und *unsatisfiableDiv*: Die Elemente mit dem Identifier *tautologyDiv*, *satisfiableDiv*, *unsatisfiableDiv* dient dazu die Prüfungsergebnisse einzublenden.
- *tautologySpan*, *satisfiableSpan* und *unsatisfiableSpan*: Die Element mit dem Identifier *tautologySpan*, *satisfiableSpan* und *unsatisfiableSpan* dient dazu die Prüfungsergebnistexte einzublenden.
- *nextStepButton*: Das Element mit dem Identifier *nextStepButton* wird genutzt, um das Button “Nächste Schritt” im Popup-Fenster “Schritt für Schritt Lösung” einzublenden.
- *stepByStepDiv*: Das Element mit dem Identifier *stepByStepDiv* wird genutzt, um die Lösung im Popup-Fenster “Schritt für Schritt Lösung” einzublenden.

TableauView kann mittels folgender Methoden den DOM-Tree manipulieren:

- *checkTautology()*: Mittels der Methode *toDisplayInput()* des Controllers enthält sie die formatierende Eingabeformel, die in das DIV-Element mit der Id *formulaInput* des DOM-Trees eingeblendet wird. Sie ruft die Methode *tautology()* des Controllers auf um das Tableau und das Ergebnis der Tautologie-Prüfung einzublenden. Bei fehlerhafter Eingaben gibt sie eine Fehlermeldung aus. Diese Fehlermeldung wird in das DIV-Element mit der Id *errorInput* des DOM-Trees eingeblendet.

```

1  TableauView.prototype.checkTautology = function () {
2      this.clearStepByStep();
3      var input = this.controller.toDisplayInput(this.input.value);
4      this.panelInput.style.display = 'block';
5      this.formulaInput.innerHTML = input;
6      google.charts.load('current', {packages: ["orgchart"]});
7      if (!InputError) {
8          this.errorInput.innerHTML = "";
9          this.controller.tautology();
10     } else {
11         var message = "";
12         for(var i = 0; i < ErrorMessage.length; i++){

```



```

13         message += ErrorMessage[i] + ". "
14     }
15     this.errorInput.innerHTML = "Fehlerhafte Eingabe!\n" + this.
translateMessage(message);
16     this.resetPage();
17 }
18 InputError = false;
19 };

```

Listing 6.23. checkTautology()(Klasse TableauView)

- *checkSatisfiable()*: Analog wie *checkTautology()*, ruft sie die Methode *satisfiable()* des Controllers aus, um das Ergebnis der Erfüllbarkeit-Prüfung auszugeben.
- *createSolutionStepChartP()*: Erhält ein HTML-Element und erzeugt in dem Element ein P-Element mit der Id, der mit “chart-step-” beginnt und dann mit der Ordnungszahl des Schrittes gefolgt wird. Dieses Element wird verwendet um der “Chart” (Abbildung 6.9) eines Schrittes von der “Schritt für Schritt Lösung” einzublenden.

```

1 TableauView.prototype.createSolutionStepChartP = function (parentElement) {
2     var p = document.createElement("p");
3     p.id = "chart-step-" + stepNr;
4     p.className = "scroll";
5     parentElement.appendChild(p);
6 };

```

Listing 6.24. createSolutionStepChartP(Klasse TableauView)

- *createSolutionStepDiv()*: Mit Hilfe der Methoden *createSolutionStepHeader()*, *createSolutionStepBody()*, *createSolutionStepChartP()* erzeugt sie ein DIV-Element mit der Id, der mit “div-step-” beginnt und dann mit der Ordnungszahl des Schrittes gefolgt wird, um ein Schritt der “Schritt für Schritt Lösung” zu erstellen. Jeder Schritt enthält die Schritt-Nummer (“Header”) die verwendete Regel (“Body”), das Diagramm (“Chart”) und den Hinweis (falls es einen gibt) (Abbildung 6.9). Diese Methode erhält einen Text, um diesen an die Methode *createSolutionStepBody()* weiterzuleiten.

```

1 TableauView.prototype.createSolutionStepDiv = function (text) {
2     var divCurrentStep = document.createElement("div");
3     divCurrentStep.id = "div-step-" + stepNr;
4     if (divCurrentStep.id != "div-step-1") {
5         var pDivider = document.createElement("p");
6         pDivider.className = "divider";
7         divCurrentStep.appendChild(pDivider);
8         divCurrentStep.style.display = 'none';
9     }
10    this.createSolutionStepHeader(divCurrentStep);
11    this.createSolutionStepBody(divCurrentStep, text);
12    this.createSolutionStepChartP(divCurrentStep);
13    this.stepByStepDiv.appendChild(divCurrentStep);
14    google.charts.setOnLoadCallback(drawTableauChartStepByStep);
15    stepNr++;
16 };

```

Listing 6.25. createSolutionStepDiv(Klasse TableauView)

- *createStepByStepSolution()*: Mit Hilfe der Methode *createSolutionStepDiv()* erstellt sie die vollständige “Schritt für Schritt Lösung” Struktur. Die Tableau-Knoten werden durch Breitensuche sortiert. Die Anzahl der Schritte entspricht der Anzahl der Knotengruppen. Jede Knotengruppe enthält nur die Knoten, die

denselben Elternknoten haben. Die Reihenfolge der Schritte entspricht die Reihenfolge der Knoten.

- *displayNextSolutionStep()*: Blendet das DIV-Element des nächsten Schrittes ein, wenn es existiert. Dieses Element wird mit der Id, die mit “div-step-” beginnt und dann mit der Ordnungszahl des nächsten Schrittes gefolgt wird.

```

1  TableauView.prototype.displayNextSolutionStep = function () {
2      var nextDivId = "div-step-" + nextStepNr;
3      var isNextDivIdExist = document.getElementById(nextDivId);
4      if (isNextDivIdExist) {
5          document.getElementById(nextDivId).style.display = "block";
6      }
7      var afterNextDivId = "div-step-" + (nextStepNr + 1);
8      var isAfterNextDivIdExist = document.getElementById(afterNextDivId);
9      if (!isAfterNextDivIdExist) {
10         this.nextStepButton.style.display = "none";
11     }
12     nextStepNr++;
13 };

```

Listing 6.26. displayNextSolutionStep(Klasse TableauView)

TableauView bietet auch die folgende Methoden:

- *hideHtmlElement()*: Erhält ein HTML-Element und blendet es aus.
- *clearStepByStepDiv()*: Setzt die globale Variablen, die für die “Schritt für Schritt Lösung” benötigt werden wie *stepNr*, *nextStepNr*, *chartNr*, *ArrOfTableauNodesByStepByStep*, *ArrOfAllTableauNodes* zurück und blendet den Button “Nächste Schritt” aus. Außerdem entfernt sie auch alle untergeordneten Knoten des *stepByStepDiv* Elements.
- *resetPage()*: Mittels Methode *hideHtmlElement()* und *clearStepByStepDiv()* blendet sie die HTML-Elemente *panel-result*, *panel-tableau*, *panel-formula* aus und ruft sie die Methode *clearStepByStepDiv()*. Weiterhin löscht sie auch die gespeicherten Fehlermeldungen.
- *translateMessage()*: Erhält eine ANTLR-Fehlermeldung und übersetzt sie ins Deutsch.

Wenn eine Instanz der Klasse *TableauView* initialisiert wird, initialisiert er auch gleich eine Controller-Instanz. Dieser Controller kann sich hierzu folgender Methoden und Attribute der *TableauView* bedienen, um den DOM-Tree manipulieren zu können:

- *createSolutionStepHeaderP()*: Erhält ein HTML-Element und erzeugt in dem Element ein P-Element um die “Header” (Abbildung 6.9) eines Schritt von der “Schritt für Schritt Lösung” einzublenden.
- *createSolutionStepBodyP()* Erhält ein Text, ein HTML-Element und erzeugt in dem Element ein P-Element um die “Body” (Abbildung 6.9) eines Schrittes von der “Schritt für Schritt Lösung” einzublenden.

Die Abbildung 6.9 erläutert den Aufbau des “Schritt für Schritt Lösung” Panel.

```

1 TableauView.prototype.createSolutionStepBody = function (parentElement, text) {
2   var p = document.createElement("p");
3   p.innerHTML = text;
4   parentElement.appendChild(p);
5 };

```

Listing 6.27. createSolutionStepBody() (Klasse TableauView)

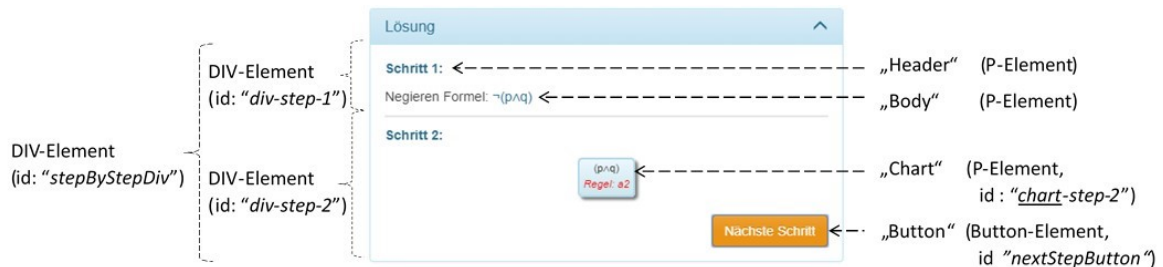


Abb. 6.9. Aufbau der “Schritt für Schritt Lösung” Panel

Das Konzept für die “Schritt für Schritt Lösung” ist es, dass zuerst die vollständige Lösung mit allen Schritten erstellt und je nach Bestätigung der “Nächste Schritt” Button wird die Lösung teilweise eingeblendet.

6.4.2 Controller

Controller (Klasse *TableauController*) sind verantwortlich für die Steuerung der Anwendung durch den Benutzer und dieser wird für die Kommunikation zwischen den Model und View verwendet. Sie werten die Eingabedaten aus und leiten sie weiter. Änderungen der Modelldaten werden also vom Controller eingeleitet. View und Controller bilden zusammen die Benutzungsoberfläche. Das Klassendiagramm der *TableauController* ist in Abbildung 6.8 gezeigt.

```

1 function TableauController(TableauView) {
2   this.view = TableauView;
3   this.rootFormula = this.getRootOfParserTree(this.view.input.value);
4   this.tableau = null;
5   this.rootTableau = null;
6   if (!InputError) {
7     this.tableau = new TableauForPropositionalLogic(this.rootFormula);
8     this.rootTableau = this.tableau.rootNode;
9   }
10  return this;
11 }

```

Listing 6.28. TableauController Konstruktor

Im Konstruktor des Controllers wird zunächst seine Methode *getRootOfParserTree()* aufgerufen. Diese Methode erhält die Eingabe, die im Attribut *input* der *TableauView* gekapselt wird, und führt die generierten Lexer und Parser aus um einen Parse-tree zu erhalten. Mit Hilfe von einer Instanz der Klasse *FormulaListener* besucht sie diesen Parse-tree, um eine Baumdarstellung der Eingabeformel zu erstellen. Dieser Baum wird im Attribut *rootFormula* des Controller gespeichert. Danach wird eine Instanz der Klasse *TableauForPropositionalLogic* initialisiert, um ein Tableau der Eingabeformel zu erstellen. Die Wurzel des Tableau wird im Attribut *rootTableau* des Controller gespeichert.

```

1 TableauController.prototype.getRootOfParserTree = function (inputString) {
2     var errorListener = new ErrorListener();
3     var chars = new antlr4.InputStream(inputString);
4     var lexer = new FormulaGrammarLexer(chars);
5     var tokens = new antlr4.CommonTokenStream(lexer);
6     var parser = new FormulaGrammarParser(tokens);
7     parser.buildParseTrees = true;
8     var tree = parser.stat();
9     var listener = new FormulaListener();
10    antlr4.tree.ParseTreeWalker.DEFAULT.walk(listener, tree);
11    var root = listener.stack.slice(-1).pop();
12    return root;
13 };

```

Listing 6.29. `getRootOfParserTree()` (Klasse `TableauController`)

Wenn der Anwender den Button “Erfüllbarkeit” betätigt, wird die Methode *satisfiable()* des Controllers ausgeführt. Die Methode wird zuerst die Eingabeformel mittels Google Charts darstellen und die Panels der View einblenden und das Prüfergebnis, das man mittels Methode *isSatisfiable()* der Klasse *TableauForPropositionalLogic* erhält, in der GUI ausgeben. Anschließend wird sie die Tableau-Darstellung einblenden. Abschließend, mit Hilfe der Methoden *createSolutionStepHeaderP()*, *createSolutionStepBodyP()* der Klasse *TableauView* wird sie die “Schritt für Schritt Lösung” erstellen.

Wenn der Anwender den Button “Tautologie” betätigt, wird die Methode *tautology()* des Controllers ausgeführt. Diese Methode analog zur Methode *satisfiable()*, dann wird sie zuerst die Eingabeformel negieren und das Attribut *rootFormula* mit der Negationsformel aktualisieren. Danach prüft sie die Erfüllbarkeit der negierten Formel und die Ergebnisse werden eingeblendet.

```

1 TableauController.prototype.tautology = function () {
2     // this.rootFormula = this.getRootOfParserTree(this.view.input.value);
3     google.charts.setOnLoadCallback(drawFormulaChart);
4     this.view.panelFormula.style.display = 'block';
5     var formula = this.rootFormula.toFormulaString();
6     this.rootFormula = this.rootFormula.getNegativeFormula();
7     this.tableau = new TableauForPropositionalLogic(this.rootFormula);
8     this.rootTableau = this.tableau.rootNode;
9     this.view.panelResult.style.display = 'block';
10    this.view.tautologyDiv.style.display = 'block';
11    this.view.unsatisfiableDiv.style.display = 'none';
12    this.view.satisfiableDiv.style.display = 'none';
13    if (this.tableau.isSatisfiable()) {
14        isTautology = TautologyOrSatisfiableEnum.FALSE;
15        isSatisfiable = TautologyOrSatisfiableEnum.NOTTESTED;
16        this.view.tautologySpan.innerHTML = formula + " ist keine ";
17    } else {
18        isTautology = TautologyOrSatisfiableEnum.TRUE;
19        isSatisfiable = TautologyOrSatisfiableEnum.NOTTESTED;
20        this.view.tautologySpan.innerHTML = formula + " ist eine ";
21    }
22    this.view.formula.innerHTML = "Negierte Formel: ";
23    this.view.tableau.innerHTML = "Tableau der negierten Formel: ";
24    this.view.panelTableau.style.display = 'block';
25    google.charts.setOnLoadCallback(drawTableauChart);
26    this.view.createSolutionStepHeaderP(this.view.stepByStepDiv);
27    var text = "Negieren Formel: <span class='text-info'>" + this.rootFormula.toFormulaString() + "</span>";
28    this.view.createSolutionStepBodyP(this.view.stepByStepDiv, text);
29    chartNr = 2;
30    stepNr++;
31 };

```

Listing 6.30. `tautology()` (Klasse `TableauController`)

Weiterhin bietet die Klasse *TableauController* noch die Methode *toDisplayInput()*. Mit Hilfe des Lexers von ANTLR formatiert die Methode die Eingabeformel (z.B von $p \& q$ nach $p \wedge q$). Diese formatierende Formel wird mittels der Klassen *TableauView* eingeblendet.

7

Softwaretest

Der Softwaretest umfasst eine Nachweisstrategie und die Vorgaben der Entwicklung, um eine hochwertige Anwendung zu erstellen.

Die Nachweisstrategie beschreibt die Anwendung von Unit-Test auf das Gesamtsystem, insbesondere die Festlegung des Einsatzes und das Code Coverage.

Vorgaben für die Entwicklungsumgebung umfasst die Konfiguration der Unit-Tests/Code Coverage Tools und die Verwendung der Testergebnisse.

7.1 Konzept zum Softwaretest

Das Konzept umfasst das Testverfahren während der Entwicklung mit abschließender Testdokumentation.

Folgende Anforderungen stellt das Konzept an das Testverfahren

- Jede Methode wird mit Unit-Tests getestet.
- Die Unit-Tests werden begleitend zur Entwicklungsphase geschrieben und erweitert
- Abschließend decken die Unit-Tests mindestens 85% der implementierten Methoden ab, Frameworks ausgenommen
- Regelmäßiges Prüfen der Code Coverage

Während des Entwicklungsprozesses wurden die Unit-Test kontinuierlich vervollständigt und mittels Code Coverage geprüft, ob die Testabdeckung ausreichend ist.

Die Unit-Tests wurden mit Hilfe von Jasmine erstellt. Mittels zwei HTML-Dateien "*UnitTestModel.html*" und "*UnitTestControllerView.html*" werden die Tests integriert. Darin erhält "*UnitTestModel.html*" die Tests für das Model. Und "*UnitTestControllerView.html*" enthält die Tests für den Controller und die View sowie einer Kopie der

indexTableau.html Dateien um die Manipulation des DOM-Tree zu testen. In der Datei *UnitTestModel.html* wird das Test-Framework Jasmine im head-Tag eingebunden (Listing 7.1). Dagegen wird Jasmine in der Datei *UnitTestControllerView.html* am Ende der Seite im body-Tag eingebunden, da es notwendig ist, dass das Markup vollständig geladen ist, bevor mit den Test begonnen werden kann (Listing 7.2).

```

1 <script src="lib/jasmine-3.1.0/jasmine.js"></script>
2 [...]
3 <script src="lib/jasmine-3.1.0/jasmine-html.js"></script>
4 <script src="lib/jasmine-3.1.0/boot.js"></script>
5
6 <!--incl. Test files here...-->
7 <script src="Tests/FormulaListenerUnitTest.js"></script>
8 <script src="Tests/FormulaUnitTest.js"></script>
9 <script src="Tests/TableauNodeUnitTest.js"></script>
10 <script src="Tests/TableauForPropositionalLogicUnitTest.js"></script>
11 </head>
12 <body>
13 </body>
14 </html>

```

Listing 7.1. Einbindung Jasmine und Test-Dateien (Model)

```

1 [...]
2 <!--incl. Test files here...-->
3 <script src="src/Global.js"></script>
4 <script src="src/TableauClient.js"></script>
5 <script src="Tests/TableauControllerAndViewUnitTest.js"></script>
6 </body>

```

Listing 7.2. Einbindung Jasmine und Test-Dateien (Controller und View)

Der Aufbau von Tests sieht wie folgt aus (Listing 7.3):

```

1 describe("Unit Tests for Formula", function() {
2   var left = new Formula("p", null, null, FormulaTypeEnum.POSITIVLITERAL);
3   var right = new Formula("p", null, null, FormulaTypeEnum.POSITIVLITERAL);
4   var root = new Formula(FormulaOperatorEnum.AND, left, right, FormulaTypeEnum.ALPHA);
5   var negation = new Formula(FormulaOperatorEnum.NEG, null, root, FormulaTypeEnum.BETA);
6
7   it("p&q has type as ALPHA, p and p have types as POSITIVLITERAL", function() {
8     expect(left.getFormulaType()).toEqual(FormulaTypeEnum.POSITIVLITERAL);
9     expect(right.getFormulaType()).toEqual(FormulaTypeEnum.POSITIVLITERAL);
10    expect(root.getFormulaType()).toEqual(FormulaTypeEnum.ALPHA);
11  });
12
13  it("p&q has left child as p and right child as p", function() {
14    expect(root.getLeft()).toEqual(left);
15    expect(root.getRight()).toEqual(right);
16  });
17
18  it("p has negative formula as !p", function() {
19    expect(left.getNegativeFormula()).toEqual(new Formula(FormulaOperatorEnum.NEG,
20    null, p, FormulaTypeEnum.NEGATIVLITERAL));
21  });
22  [...]
23 });

```

Listing 7.3. Beispiel Test der Klasse Formula

7.2 Evaluation der Testfälle

Es gibt insgesamt 72 Unit-Tests für das Model (Abbildung 7.1) und 16 Unit-Tests für den Controller und die View (Abbildung 7.2).



The screenshot shows the Jasmine test runner interface. At the top, it says "Jasmine 3.1.0" and "Options". Below this, a green bar indicates "72 Testss, 0 failures, randomized with seed 81765" and "finished in 0.339s". The main content area displays the following test results:

```

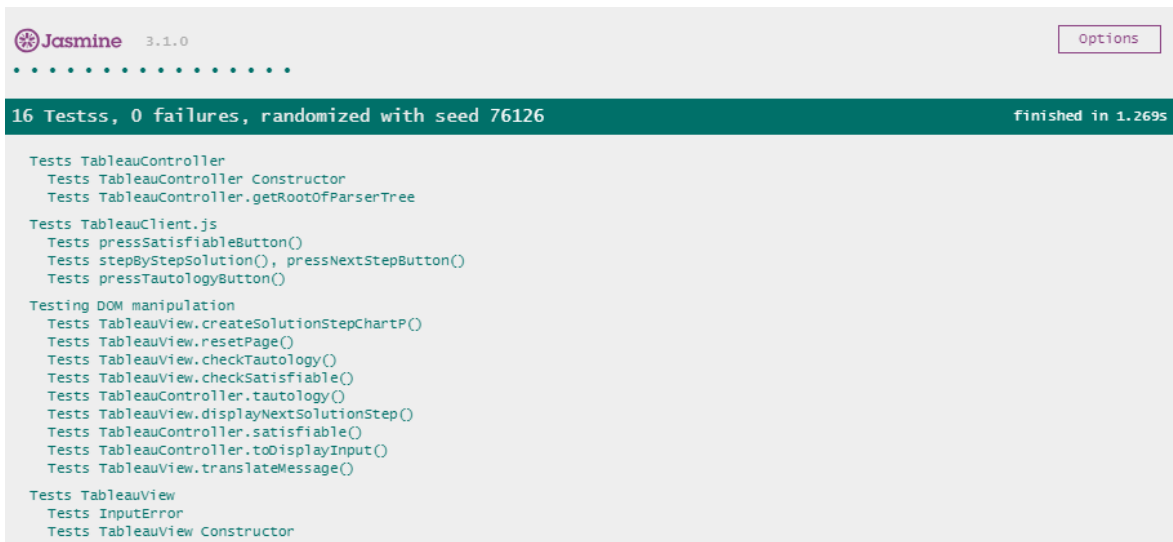
Unit Tests for Class Formula
  in.order traversal of p is [p]
  root has string as "p ^ p"
  p&q has type as ALPHA, p and q have types as POSITIVLITERAL
  in.order traversal of p&p hat 3 elements
  0 has negative formula as !1
  p&q has negative formula as !(p&q)
  root has label as " ^ " and the label of left is the same the label of right
  p&q has left child as p and right child as q
  !p has negative formula as p
  1 has negative formula as !0
  p has negative formula as !p
  test getFormulasForChart()

Unit Tests for Class FormulaListener

Tests α-formula
  !(p!&q) is a α-formula
  !!!(p&q) is a α-formula
  p!|q is a α-formula
  !!!((p|q)&(p&q)) is a α-formula
  p<=>q is a α-formula
  !(p^q) is a α-formula
  !!q is a α-formula
  p&q is a α-formula
  !(p|q) is a α-formula
  !!(p|q) is a α-formula
  !(p=>q) is a α-formula

```

Abb. 7.1. Ein Teil der Testergebnisse des Models



The screenshot shows the Jasmine test runner interface. At the top, it says "Jasmine 3.1.0" and "Options". Below this, a green bar indicates "16 Testss, 0 failures, randomized with seed 76126" and "finished in 1.269s". The main content area displays the following test results:

```

Tests TableauController
  Tests TableauController Constructor
  Tests TableauController.getRootOfParserTree

Tests TableauClient.js
  Tests pressSatisfiableButton()
  Tests stepByStepSolution(), pressNextStepButton()
  Tests pressTautologyButton()

Testing DOM manipulation
  Tests TableauView.createSolutionStepChartP()
  Tests TableauView.resetPage()
  Tests TableauView.checkTautology()
  Tests TableauView.checkSatisfiable()
  Tests TableauController.tautology()
  Tests TableauView.displayNextSolutionStep()
  Tests TableauController.satisfiable()
  Tests TableauController.toDisplayInput()
  Tests TableauView.translateMessage()

Tests TableauView
  Tests InputError
  Tests TableauView Constructor

```

Abb. 7.2. Testergebnisse des Controllers und der View

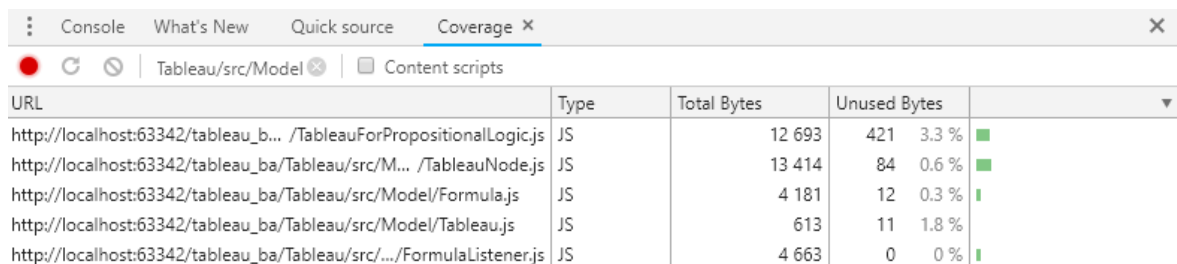
Die Tests sind nach der Klasse gruppiert und können dementsprechend zielgerichtet evaluiert werden.

- *FormulaUnitTest.js*: Enthält 12 Tests für die Klasse *Formula*, wobei alle Methoden der Klasse *Formula* mindestens einmal überprüft werden. Die Methode *getNegativeFormula()* wird mit den Eingabeformeln als Konstante True/False, positive-/negative-Literal, α - und β -Formel überprüft.
- *FormulaListenerUnitTest.js*: Enthält 30 Tests. Diese Tests untersuchen die Struktur einer Formel wie linken (*left*) und rechten (*right*) Teilbaum, inhalt (*label*) und Formeltypen (*formulaType*) nachdem sie mithilfe der Klasse *FormulaListener* von Zeichenkette in einen Baum konvertiert wurden. Für jeden Formeltyp (Konstante True/False, positive-/negative-Literal, α - und β -Formel) und für jede Regel der

Klassifizierung von α - und β -Formeln wird mindestens eine Formel überprüft.

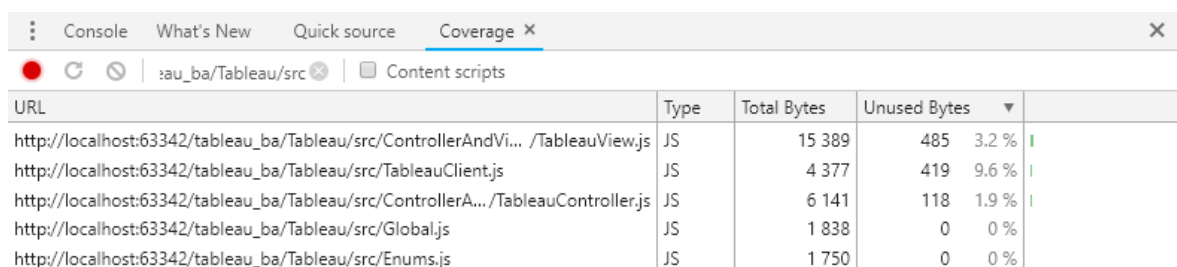
- *TableauNodeUnitTest.js*: Enthält 12 Tests für die Klasse *TableauNode*, wobei alle Methoden der Klasse *TableauNode* mindestens einmal überprüft werden. Alle Methoden, die den booleschen Wert zurückgeben, werden mindestens zweimal mit den Rückgabewerten false und true überprüft. Die Methoden *containsOnlyLiterals()*, *containsAComplementaryPairOfLiterals()* und *containsAlphaFormula()* werden mit den Tableau-Knoten, die mit der Menge von Formeln $\{q\}$, $\{p \wedge (q \vee \neg p)\}$, $\{p, q\}$, $\{p, \neg p\}$, $\{\neg p, p\}$, $\{p, q \vee \neg p\}$, $\{p, \neg p, \neg \neg p\}$, $\{p, 1\}$ und $\{q, 0\}$ markiert sind, getestet.
- *TableauForPropositionalLogicUnitTest.js*: Enthält 18 Tests für die Klasse *TableauForPropositionalLogic*, wobei alle Methoden der Klasse *TableauForPropositionalLogic* mindestens einmal überprüft werden. Die Methoden *determineAlphaFormulas()* und *determineBetaFormulas()* werden für jede Regel der Klassifizierung von α - und β -Formeln mindestens einmal überprüft.
- *TableauControllerAndViewUnitTest.js*: Enthält 16 Tests für die Klassen *TableauView*, *TableauController*, *TableauClient.js* und *TableauClient.js* wobei alle Methoden mindestens einmal überprüft werden. Die Methoden *pressSatisfiableButton()* und *pressTautologyButton()* werden für eine syntaktisch korrekte Formel und eine fehlerhafte Formel einmal überprüft.

7.3 Code Coverage



URL	Type	Total Bytes	Unused Bytes	
http://localhost:63342/tableau_b... /TableauForPropositionalLogic.js	JS	12 693	421 3.3 %	■
http://localhost:63342/tableau_ba/Tableau/src/M... /TableauNode.js	JS	13 414	84 0.6 %	■
http://localhost:63342/tableau_ba/Tableau/src/Model/Formula.js	JS	4 181	12 0.3 %	■
http://localhost:63342/tableau_ba/Tableau/src/Model/Tableau.js	JS	613	11 1.8 %	■
http://localhost:63342/tableau_ba/Tableau/src/.../FormulaListener.js	JS	4 663	0 0 %	■

Abb. 7.3. Code Coverages des Models



URL	Type	Total Bytes	Unused Bytes	
http://localhost:63342/tableau_ba/Tableau/src/ControllerAndVi... /TableauView.js	JS	15 389	485 3.2 %	■
http://localhost:63342/tableau_ba/Tableau/src/TableauClient.js	JS	4 377	419 9.6 %	■
http://localhost:63342/tableau_ba/Tableau/src/ControllerA... /TableauController.js	JS	6 141	118 1.9 %	■
http://localhost:63342/tableau_ba/Tableau/src/Global.js	JS	1 838	0 0 %	■
http://localhost:63342/tableau_ba/Tableau/src/Enums.js	JS	1 750	0 0 %	■

Abb. 7.4. Code Coverages des Controllers und der View

Um eine Code Coverage zu messen, ist ein weiteres Tool notwendig. Es gibt dafür Chrome DevTools, das besteht aus einer Reihe von Webentwicklungstools, die direkt in den Google Chrome-Browser integriert sind. Die neueste DevTools Version (Chrome 59) bietet eine neue Registerkarte "Coverage" an. Wenn man eine Seite lädt oder

ausführt, erfährt man auf dieser Registerkarte, wie viel Prozent des heruntergeladenen Codes, noch nicht verwendet wurde [Bas18]. Die Ergebnisse der Codeabdeckung sind unter den Abbildungen 7.3 und 7.4 zu sehen.

Alle Dateien erfüllen die Mindestgrenze unter Rücksichtnahme, dass das Tool bei *switch-case* und *if-clause*(ohne *else*) nicht richtig ausgewertet werden konnte. Außerdem konnte das Tool auch nicht jedes Event des GUI mit auswerten, deshalb kommt es zu verfälschten Ergebnissen (Abbildung 7.5).

```

149      switch (alphaFormula.getRight().getLabel()) {
150          case FormulaOperatorEnum.NEG: { //neg
151              alpha1Formula = null;
152              alpha2Formula = alphaFormula.getRight().getRight();
153              alphaRule = RuleEnum.ALPHA_1;
154              break;
155          }
156          case FormulaOperatorEnum.OR: { //or
157              var formulaTypLeft = alphaFormula.getRight().getLeft().getFormulaTypAfterNegation();
158              alpha1Formula = new Formula(FormulaOperatorEnum.NEG, null, alphaFormula.getRight().getLeft(), formulaTypLeft);
159              var formulaTypRight = alphaFormula.getRight().getRight().getFormulaTypAfterNegation();
160              alpha2Formula = new Formula(FormulaOperatorEnum.NEG, null, alphaFormula.getRight().getRight(), formulaTypRight);
161              alphaRule = RuleEnum.ALPHA_3;
162              break;
163          }
164          case FormulaOperatorEnum.NAND: { //nand
165              alpha1Formula = alphaFormula.getRight().getLeft();
166              alpha2Formula = alphaFormula.getRight().getRight();
167              alphaRule = RuleEnum.ALPHA_5;
168              break;
169          }
170          case FormulaOperatorEnum.IMP: { //imp

```

Abb. 7.5. Problem bei Code Coverages

8

Zusammenfassung

Diese Arbeit hat die beschriebenen Anforderungen erfüllt. Die Kernaufgabe besteht darin, eine Webanwendung für ein semantisches Tableau zu entwickeln, um ermitteln zu können, ob eine Formel erfüllbar oder allgemeingültig ist. Darüber hinaus kann die Anwendung die Struktur von einem Tableau über Google Chart Tools visualisieren. Zur Unterstützung bei dem Erlernen der Konstruktion eines semantischen Tableaus bietet die Anwendung auch eine “Schritt für Schritt Lösung” -Funktion, wonach Studierende Schritt für Schritt den Aufbau eines Tableaus folgen können. Nicht nur das, die Anwendung wurde entwickelt, um eine Vielzahl von Geräten von Desktop bis Smartphone zu unterstützen.

Es wäre ideal, wenn die Anwendung in Zukunft weiterentwickelt werden könnte, zum Beispiel: Unterstützung für die prädikatenlogischen Formeln oder Verwendung eines Offline-Frameworks, um ein Tableau ohne Internetverbindung visualisieren zu können.

Danksagung

Ich möchte mich recht herzlich bei Herrn Prof. Dr. rer. nat. Andreas Schäfer bedanken, der stets für mich ansprechbar war sowie mit viel Engagement meine Arbeit betreut hat.

Ebenfalls herzlich danken möchte ich Herrn Prof. Dr.-Ing. Stefan Krause, der sich kurzfristig bereit erklärte, als Zweitprüfer zur Verfügung zu stehen.

Abbildungsverzeichnis

2.1	Anwendungsfalldiagramm	4
3.1	Beispiel Konstruktion für eine Formel	10
3.2	Beispiel Konstruktion für semantische Tableaus	13
3.3	Ein weiteres Tableau	14
4.1	Desktop-First und Mobile-First	21
4.2	Unterstützte Browsers von Bootstrap	22
4.3	Beispiel Grid System	23
4.4	Jasmine-Bibliotheksdateien	29
5.1	Achitektur	31
5.2	Klassendiagramm	32
6.1	Ordnerstruktur	35
6.2	Ordnerstruktur antlr4GeneratedSources	36
6.3	Ordnerstruktur lib	36
6.4	Ordnerstruktur src	36
6.5	Ordnerstruktur test	37
6.6	KlassendiagrammModel	39
6.7	Screenshots der Tableau-Anwendung	49
6.8	KlassendiagrammControllerView	51
6.9	Aufbau der StepByStepPanel	55
7.1	Testergebnisse des Models	60
7.2	Testergebnisse des Controllers und der View	60

7.3	Code Coverages des Models	61
7.4	Code Coverages des Controllers und der View	61
7.5	Problem bei CodeCoverages	62

Tabellenverzeichnis

3.1	Notationen der Booleschen Operatoren	11
3.2	Klassifizierung von α - und β -Formeln	14
4.1	Rasteroptionen im Bootstrap-Framework	24

Literatur

- [BA12] BEN-ARI, Mordechai: *Mathematical Logic for Computer Science*. 3.rd. London : Springer-Verlag, 2012
- [Bas18] BASQUES, Kayce: *What's New In DevTools (Chrome 59)*, ges. am 11.08.2018. <https://developers.google.com/web/updates/2017/04/devtools-release-notes>. Version: 2018
- [Boo18a] BOOTSTRAP: *Bootstrap*, ges. am 9.08.2018. <https://getbootstrap.com/docs/4.1/about/overview/>. Version: 2018
- [Boo18b] BOOTSTRAPWORLD: *Was ist Bootstrap?*, ges. am 9.08.2018. <https://www.bootstrapworld.de/was-ist-bootstrap.html>. Version: 2018
- [Cha18] CHARTS, Google: *About Google chart tools*, ges. am 10.08.2018. <https://developers.google.com/chart/>. Version: 2018
- [Dre18] DREISEITL, Stephan: *Mathematik für Software Engineering*. Berlin : Springer Vieweg, 2018
- [Gon18] GONZALO, Frederic: *Understanding the difference between mobile-first, adaptive and responsive design*, ges. am 9.08.2018. <http://fredericgonzalo.com/en/2017/03/01/understanding-the-difference-between-mobile-first-adaptive-and-responsive-design/>. Version: 2018
- [Goo18a] GOOGLE: *Draw the Chart*, ges. am 10.08.2018. https://developers.google.com/chart/interactive/docs/basic_draw_chart. Version: 2018
- [Goo18b] GOOGLE: *Organization Chart*, ges. am 10.08.2018. <https://developers.google.com/chart/interactive/docs/gallery/orgchart>. Version: 2018
- [Jas18] JASMINE: *Draw the Chart*, ges. am 11.08.2018. <https://jasmine.github.io/>. Version: 2018
- [Kro16] KROHN, Uwe: *Implementierung von binären Bäumen. Skript zur Vorlesung Informatik 2*. Lübeck, 2016
- [kul18] KULTURBANAUSE: *Was bedeutet Desktop First?*, ges. am 9.08.2018. <https://kulturbanause.de/faq/desktop-first/>. Version: 2018
- [Par12] PARR, Terence: *The Definitive ANTLR 4 Reference*. Dallas, Texas : Pragmatic Bookshelf, 2012

- [Pix12] PIXELS|BYTES: *Node.js require() for your browser*, ges. am 11.08.2018. <https://pixelsvsbytes.com/2012/04/node-js-require-for-your-browser/?phpMyAdmin=0ZRJj6JUwx3do5nBBGsib-MdEm7>. Version: 2012
- [Poi18] POINT, Tutorials: *JasmineJS - Overview*, ges. am 11.08.2018. https://www.tutorialspoint.com/jasminejs/jasminejs_overview.htm. Version: 2018
- [Sch13] SCHENKE, Michael: *Logikkalküle in der Informatik : Wie wird Logik vom Rechner genutzt?* Wiesbaden : Springer Vieweg, 2013
- [Sch17] SCHÄFER, Andreas: *Informatik 1. Skript zur Vorlesung*. Lübeck, 2017
- [Sta18] STATCOUNTER: *Desktop vs Mobile vs Tablet Market Share Worldwide - July 2018*, ges. am 9.08.2018. <http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet>. Version: 2018
- [Wik18a] WIKIPEDIA: *Bootstrap (Framework)*, ges. am 9.08.2018. [https://de.wikipedia.org/wiki/Bootstrap_\(Framework\)](https://de.wikipedia.org/wiki/Bootstrap_(Framework)). Version: 2018
- [Wik18b] WIKIPEDIA: *Google Charts*, ges. am 10.08.2018. https://en.wikipedia.org/wiki/Google_Charts. Version: 2018