Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Julia Schmaglowski

Entwicklung eines Filters für valide Erweiterungen von unvollständigen Syntaxbäumen zur Verbesserung der UX im Rahmen der Lernumgebung BlattWerkzeug

Fakultät Design, Medien und Information
Studiendepartment Medientechnik

Faculty of Design, Media and Information
Department of Media Technology

Julia Schmaglowski

Entwicklung eines Filters für valide Erweiterungen von unvollständigen Syntaxbäumen zur Verbesserung der UX im Rahmen der Lernumgebung BlattWerkzeug

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Media Systems
am Department Medientechnik
der Fakultät Design, Medien und Information
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Larissa Putzar
Zweitgutachter: André Jeworutzki

Eingereicht am: 29. Januar 2025

Julia Schmaglowski

Thema der Arbeit
Entwicklung eines Filters für valide Erweiterungen von unvollständigen Syntaxbäumen zur Verbesserung der UX im Rahmen der Lernumgebung BlattWerkzeug

Stichworte
UX, Syntaxfreie Programmierung, Lernsoftware

Kurzzusammenfassung
Diese Bachelorarbeit befasst sich mit der Konzeption und Entwicklung eines Filters für valide Erweiterungen von unvollständigen Syntaxbäumen in der Lernumgebung BlattWerkzeug. Dieser Filter soll die User Experience bei Verwendung des Drag & Drop Editors verbessern. Dazu werden in der Konzeption Prinzipien der menschenzentrischen Gestaltung und Usability, Nielsen Heuristik, Regeln für gute UX und die Eigenschaften syntaxfreier Programmierung betrachtet. Die gewonnenen Erkenntnisse ermöglichen die Erstellung von Profilen der Nutzenden und Anforderungen. Der Filter wird vollständig als Feature in das vorhandene BlattWerkzeug Projekt integriert und in Angular nach dem Observer Pattern implementiert.
Julia Schmaglowski

Title of the paper
Development of a filter for valid extensions of incomplete syntax trees to improve the UX in the context of the BlattWerkzeug learning environment

Keywords
UX, syntaxfree Programming, Educational Software

Abstract
This bachelor thesis deals with the design and development of a filter for valid extensions of incomplete syntax trees in the BlattWerkzeug learning environment. It is intended to improve the user experience when using the Drag & Drop Editor. To this end, the filterdesign is based on the principles of human-centered design and usability, Nielsen heuristics, laws for good UX and the properties of syntax-free programming. The knowledge gained from researching these enables the creation of user profiles and requirements. The filter is fully integrated as a feature in the existing BlattWerkzeug project and implemented in Angular according to the observer pattern.

Inhaltsverzeichnis

1 Einleitung

Durch die Cognitive Load Theory (CTL) ist bekannt, dass die kognitive Belastung, die im Lernprozess entsteht, durch die Gestaltung des Lernmaterials beeinflusst wird [Swe05].

Bei dieser Arbeit wird es um die Überarbeitung und Erweiterung des Block-Editors in BlattWerkzeug gehen, in dem Kinder und Jugendliche mit Hilfe von Drag & Drop Operationen graphische Blöcke zu Programmcode kombinieren können. Sie sollen so an Programmierung und programmatisches Denken herangeführt werden.

BlattWerkzeug ist eine Webanwendung, die sich als Lernumgebung in der Lücke zwischen Anwendungen mit syntaxfreien Blocksprachen und klassischen Entwicklungsumgebungen sieht. Die im Drag & Drop Editor erstellten Programme zeigen optische Ähnlichkeit mit realem Programmcode, sodass die tatsächliche Codestruktur, wenn auch durch Blöcke verschleiert, noch erkennbar ist (Abb. 1.1). Zur Zeit sind mehrere unterschiedliche Sprachen, sowie ein Editor zum Erstellen eigener Sprachen, in BlattWerkzeug implementiert. Die in dieser Arbeit entwickelten Änderungen zur Interaktion mit Fehlern im Editor sollen für alle implementierten Sprachen funktionieren, da diesen derselbe Programmcode zugrundeliegt. Diese Arbeit konzentriert sich jedoch auf Funktionen im in BlattWerkzeug implementierten SQL-Editor, welcher das Programmieren in SQL und den Umgang mit Datenbanken im Generellen ermöglicht.

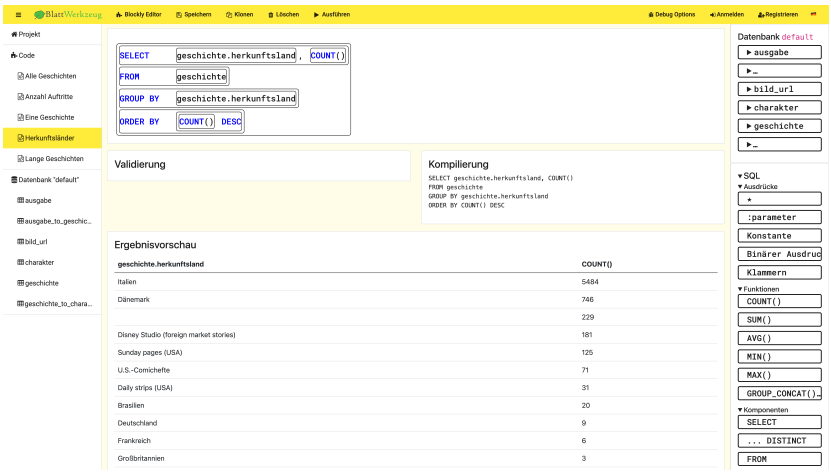

Abbildung 1.1: Bildschirmaufnahme des Drag & Drop Editors in BlattWerkzeug

1.1 Zielsetzung

Im Rahmen dieser Arbeit soll die User Experience (UX) im Umgang mit dem Block-Editor von BlattWerkzeug verbessert werden. Dazu soll vornehmlich das User Interface und das Verhalten bei der Interaktion mit Löchern im Blockcode bearbeitet werden (Löcher siehe Abb. 1.2). Ziel ist es, die nicht-lernbezogene kognitive Belastung, die benötigt wird um BlattWerkzeugs Funktionsweise zu verstehen, zu verringern, sodass die Nutzenden einer stärkeren lernbezogenen kognitiven Belastung ausgesetzt werden können. Eine positive Nutzungserfahrung, gerade im Zusammenhang mit dem Lösen von Problemen in BlattWerkzeug, hinterlässt bei Lernenden Erfolgserlebnisse.

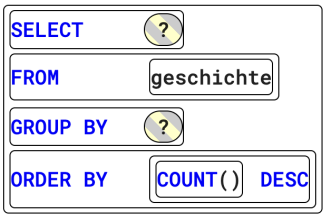

Abbildung 1.2: Blockcode mit Löchern aus BlattWerkzeug

Der Ersteindruck den Benutzende von BlattWerkzeug erhalten, kann durch die in dieser Arbeit angestrebten Verbesserungen entscheidend positiver ausfallen. Um Neulinge nicht mit der Masse an möglichen Blöcken zu überfordern, soll es möglich werden, direkter mit

den Löchern im Code zu interagieren Dies soll ermöglicht werden, indem nach einem Klick auf die im Code zu bearbeitende Stelle nur valide Blöcke für die jeweiligen Lücken in der Seitenleiste angezeigt werden. Durch diese Beschränkung der Auswahlmöglichkeiten wird auch die Arbeitsgeschwindigkeit für erfahrenere Nutzende erhöht. Beide Gruppen werden motiviert, BlattWerkzeug erneut zu Nutzen und sich über die Grenzen des vorliegenden Tools hinaus mit Programmierung zu beschäftigen. Dieser letzte Punkt geht auch mit dem ursprünglichen Grundgedanken von BlattWerkzeug, eine Brücke zwischen Beginnertools wie Scratch und dem professionellen Texteditor zu präsentieren, überein.

1.2 Aufbau der Arbeit

Diese Arbeit ist in fünf Kapitel unterteilt, die wie folgt strukturiert sind:

Kapitel 2 bietet einen Einblick in die Grundlagen, die relevant für das Verständnis der folgenden Kapitel sind. Diese Grundlagen umfassen fünf Abschnitte, darunter sind eine Einführung in die Prinzipien der menschenzentrischen Gestaltung, eine Darlegung unterschiedlicher Regeln, die bei der Entwicklung von gutem User Experience Design beachtet werden sollten,eine Auseinandersetzung mit den Nielsen Heuristiken zu Bewertung von Usability, eine Differenzierung von Low-Code und syntaxfreier Programmierung und eine genauere Einführung in das Tool BlattWerkzeug und dessen technische Grundlagen.

Kapitel 3 umfasst die Anforderungsanalyse. Diese zeigt die Zielgruppe von BlattWerkzeug mittels Personas und stellt funktionale und nicht-funktionale Anforderungen auf, die im Zuge der Arbeit umgesetzt werden sollen. Darüber hinaus wird in diesem Kapitel BlattWerkzeug genauer vorgestellt und die geplante Funktionsweise des Filters unter Beachtung der UX Regeln aufgezeigt.

Kapitel 4 dokumentiert den Prozess der Implementierung der umgesetzten Anforderungen.

Kapitel 5 zeigt anhand der Nielsen Heuristiken auf, inwiefern die User Experience verbessert werden konnte und stellt Probleme in BlattWerkzeugs Design vor, die während der Erstellung diese Arbeit deutlicher aufgefallen sind.

Kapitel 6 stellt abschließend heraus welche Anforderungen erfüllt wurden, und gibt einen Ausblick, durch welche zusätzlichen Erweiterungen des Ergebnis der Arbeit in Zukunft noch weiter verbessert werden könnte.

2 Grundlagen

Um die Basis für die weiteren Kapitel zu legen, werden zunächst die Begriffe der menschenzentrischen Gestaltung und User Experience erläutert. Anschließend werden unterschiedliche UX Regeln, die es im Zusammenhang mit der Arbeit zu beachten gilt, vorgestellt. Danach gibt es eine Gegenüberstellung von Low-Code und Syntaxfreier Programmierung, um anschließend auf die dieser Arbeit zugrundeliegende Lern-Entwicklungsumgebung BlattWerkzeug einzugehen. Abschließend findet in diesem Kapitel eine Vorstellung der technischen Grundlagen statt, die zum Verständnis dieser Arbeit vorausgesetzt sind.

2.1 Menschenzentrische Gestaltung und User Experience

Ein Leitwerk, welches auf dem Human-Centered Design (HCD)-Ansatz basiert und deswegen in dieser Arbeit Verwendung finden soll, ist die DIN EN ISO 9241 - Ergonomie der Mensch-System Interaktion. In diesem Werk werden unter anderem Anforderungen an interaktive Systeme unter dem Gesichtspunkt des HCD gestellt, außerdem bietet es eine allgemeine, jedoch systematische Herangehensweise zur Bewertung und Verbesserung der User Experience.

2.1.1 Menschenzentrische Gestaltung - HCD

Als menschenzentrische Gestaltung (en: Human-Centered Design) versteht man laut ISO 9241-210 die Herangehensweise bei der Gestaltung und Entwicklung von Systemen, die darauf abzielt, interaktive Systeme gebrauchstauglicher zu machen, indem sie sich auf die Verwendung des Systems konzentrieren und Kenntnisse und Techniken aus den Bereichen der Arbeitswissenschaften/Ergonomie und Gebrauchstauglichkeit anwendet [DIN20, S. 3.11]. Des Weiteren beschreibt ISO 9241-210 gebrauchstaugliche Systeme und Produkte in sowohl technischer, als auch kommerzieller Hinsicht als tendenziell erfolgreicher. So sinken beispielsweise die Kosten, in ökonomischer und kognitiver Hinsicht, für die Betreuung und Beratung von Kindern, wenn diese ohne zusätzliche Hilfe verstehen, wie die Software zu benutzen ist. Im Fall von BlattWerkzeug muss zum Beispiel, bei einer verbesserten Gebrauchstauglichkeit des Projekteditors, im Fehlerfall weniger durch eine dritte Person, wie Lehrpersonal, erklärt werden.

Bei der menschenzentrischen Entwicklung eines interaktiven Systems müssen während der Gestaltung vier miteinander verbundene Gestaltungsaktivitäten erfolgen:

a) Verstehen und Beschreiben des Nutzungskontexts

b) Spezifizieren der Nutzungsanforderungen

c) Erarbeiten von Gestaltungslösungen

d) Evaluieren der Gestaltung

[DIN20, S. 7.1]

Erarbeiten von Gestaltungslösungen Im Prozess der Gestaltung des Benutzererlebnisses sollte sowohl die Zufriedenstellung des Benutzenden auf emotionaler und ästhetischer Ebene, als auch Effizienz und Effektivität berücksichtigt werden. Eine Vielzahl von kreativen Ansätzen in der Gestaltung kann zu einer positiven User Experience beitragen. Die folgenden, aus ISO 9241-110 entnommenen, Gestaltungsgrundsätze stehen als Leitlinien zur Unterstützung der Gestaltungsaktivitäten zur Verfügung:

a) Aufgabenangemessenheit

b) Selbstbeschreibungsfähigkeit

c) Konformität mit Benutzererwartungen

d) Lernförderlichkeit

e) Steuerbarkeit

f) Fehlertoleranz

g) Individualisierbarkeit

[DIN19, S. 7.4.2.1]

2.1.2 Usability

Usability bzw. Gebrauchstauglichkeit konzentriert sich laut DIN EN ISO 9241-11 auf die Effektivität, Effizienz und Zufriedenstellung der Interaktion des Benutzenden mit dem Betrachtungsgegenstand.

Effektivität Unter Effektivität versteht sich die Genauigkeit und Vollständigkeit, mit der Benutzende bestimmte Ziele erreichen. Teilaspekte der Effektivität sind Genauigkeit und Vollständigkeit.

Effizienz Mit Effizienz bezeichnet man das Verhältnis zwischen erreichten Ergebnissen und eingesetzten Ressourcen, wie Zeit, menschlichem Aufwand, Geld und Materialien.

Zufriedenstellung der Interaktion Das Ausmaß, in dem physische, kognitive und emotionale Reaktionen des Benutzenden, die aus der Benutzung des Systems resultieren, mit der Benutzererwartung übereinstimmen, wird als Zufriedenstellung der Interaktion beschrieben.

An dieser Stelle ist herauszustellen, inwiefern Usability in interaktiven Systemen relevant ist. Sie befähigt neue und bestehende Benutzende des Systems, unabhängig von deren bestehenden Fähigkeiten, Ziele effektiv, effizient und zufrieden zu erreichen. Dabei werden das Risiko und die Folgen von unerwünschten Nutzungsfehlern auf ein Mindestmaß reduziert.

User Experience

User Experience umfasst die Wahrnehmungen und Reaktionen einer Person, die aus der tatsächlichen und/oder der erwarteten Benutzung eines Systems, eines Produkts oder einer Dienstleistung resultieren [DIN18, S. 3.2.3]. User Experience bezieht sich im Wesentlichen auf die Natur dieser Reaktionen vor, während und nach der Nutzung. (Abb. 2.1)

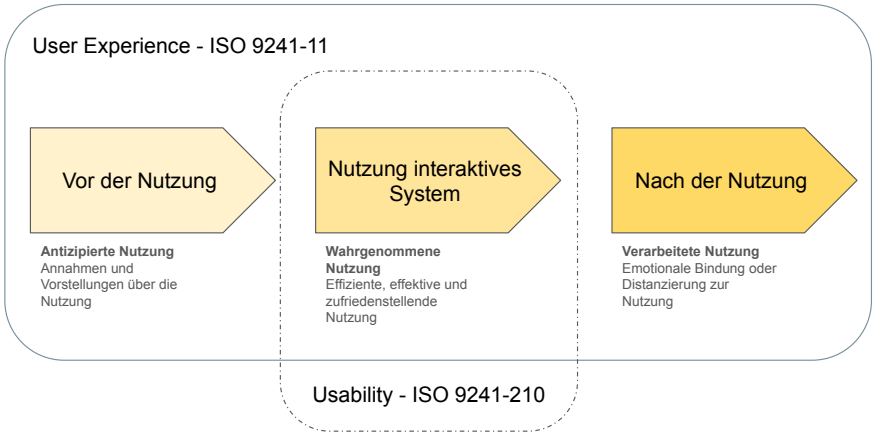

Abbildung 2.1: Definition von UX und Usability

2.2 UX Regeln

In seinem Buch "Laws of UX" konsolidiert Designer Jon Yablonski 10 psychologische Konzepte, die als Regeln für bessere UX gelten. Für diese Arbeit relevant sind besonders die Folgenden Regeln:

Tesler's Law

> Tesler's Law, also known as the law of conservation of complexity, states that for any system there is a certain amount of complexity that cannot be reduced [Yab20, S.87].

Tesler's Law befasst sich mit der Tatsache, dass es ab einem gewissen Punkt nicht mehr möglich ist die Komplexität eines Systems zu verringern. Wenn dieser Punkt erreicht ist, kann die Komplexität nur noch "verschoben" werden. So ergibt sich, dass es im Sinne von gutem UX Design ist, wenn komplexe Aufgaben möglichst vom Nutzer weg, weiter ins Backend des Systems geschoben werden. Beim Verschieben der Komplexität muss jedoch beachtet werden, dass Einfachheit nicht zu Abstraktion wird. Zu weit abstrahierte Icons für ausführbare Aktionen

können so beispielsweise eher zu Verwirrung führen, als die mentale Last zu verringern, die von der Aufgabe ausgeht.

Jakob's Law

> Users spend most of their time on other sites, and they prefer your site to work the same way as all the other sites they already know [Yab20, S.1].

2020 von Jakob Nielsen aufgestellt, besagt "Jakob's law of the internet experience", dass Nutzer*innen eine Erwartungshaltung an Webseiten entwickeln, die auf ihren kumulierten Erfahrungen mit ähnlich aussehenden Webseiten basieren. Es entsteht bei ihnen ein mentales Modell davon, wie das interaktive System funktionieren sollte. Nutzer*innen wenden dann dieses Modell wieder auf ähnliche Systeme an. Wenn man sich als Designer dieses mentale Modell zu Nutzen machen kann, dann ist mit einer Verbesserung der User Experience zu rechnen. Entspricht das Design eines Systems dem mentalen Modell des Benutzenden, kann dieser sich zum Beispiel durch Familiarität mit dem Interface eher auf die eigentlichen Zielaufgabe konzentrieren. Familiarität beim Benutzenden kann beispielsweise durch die Anwendung geläufiger Design Patterns an strategisch sinnvollen Stellen, wie Seitenstruktur, Navigation oder Platzierung von Elementen erreicht werden.

Hick's Law

> The time it takes to make a desicion increases with the number and complexity of choices [Yab20, S.23].

Aufgestellt 1952 von Psychologen William Edward Hick und Ray Heyman befasst sich Hick's Law im Kern mit dem Konzept der kognitiven oder mentalen Last. Grundsätzlich benötigen alle Aufgaben, die eine Person erledigt, einen gewissen Grad an mentalen Ressourcen. Wie hoch dieser Grad, ist hängt dabei von unterschiedlichen Kriterien ab. Ziel von guter UX ist es, die von der Benutzung der Software ausgelöste mentale Last möglichst gering zu halten, sodass Nutzende ausreichend kognitive Kapazitäten für die Bewältigung ihrer Zielaufgabe haben. Um dies zu gewährleisten, sollten die Punkte an denen Nutzende Entscheidungen treffen müssen, möglichst gering gehalten werden. Komplexe Aufgaben sollten in kleinere Schritte aufgeteilt werden. Auch bei Hick's Law sollte darauf geachtet werden, dass Einfachheit nicht gleich abstrakt bedeutet.

Peak-End Rule

> People judge an experience largely based on how they felt at its peak and at its end, rather than on the total sum or avarage of every moment of the experience [Yab20, S.53].

Die 1993 von David Kahneman aufgestellte Peak-End Rule besagt, dass Nutzende die Erfahrung mit einem interaktiven System häufig aufgrund ihrer Emotionen zum Ende und am Höhepunkt der Erfahrung beurteilen, anstatt aufgrund der Erfahrung als Gesamtes. Um sich diese Regel zu Nutzen zu machen, sollten Designer die Momente identifizieren an denen das Produkt am wertvollsten, hilfreichsten und unterhaltsamsten wahrgenommen wird. Verbesserungen an diesen Punkten können zu signifikanten Verbesserungen in der User Experience führen, da Erfahrungen, die mit starken Emotionen verknüpft sind, besser in Erinnerung bleiben.

Aesthetic-Usability Effect

> Users often percieve aesthetically pleasing design as design that's more usable [Yab20, S.65].

Die Studie "Apparent Usability vs Inherent Usability[...]" von Masaaki Kurosu und Kaori Kashimura belegt, dass eine Korrelation zwischen Ästhetik und Usability vorliegt. Nutzende sind bereit, über kleine Fehler in der Gebrauchstauglichkeit hinwegzusehen, wenn das Produkt ästhetisch ansprechend ist. Hier kann ein Zusammenhang zur Peak-End Rule herangezogen werden, denn eine positive emotionale Reaktion durch ein ansprechendes Design bleibt eher im Gedächtnis des Nutzenden.

2.3 Expertenbasierte Evaluation anhand von Nielsen Heuristiken

Im User Interface (UI) Design gelten die 1994 von Jakob Nielsen aufgestellten *10 Usability Heuristics for User Interface Design* als ganz besonders wichtig. Diese Heuristiken dienen als Leitlinien für die Beurteilung der Usability einer Anwendung und können auf den Einzelfall angewendet werden:

1: Visibility of System Status Das System sollte Nutzende stets über den aktuellen Stand der Dinge informieren.

2: Match Between the System and the Real World Das Design sollte so gestaltet sein, dass verwendete Wörter, Ausdrücke und Konzepte, dem Nutzenden vertraut sind. Wichtig ist es dabei Konventionen der realen Welt zu befolgen, um Informationen in einer natürlichen und logischen Reihenfolge erscheinen zu lassen.

3: User Control and Freedom Nutzende brauchen immer eine klare und schnelle Option um unerwünschte Aktionen Rückgängig zu machen.

4: Consistency and Standards Elemente innerhalb einer Anwendung sollte konstistent sein. Dazu zählt auch auf konventionen der Plattform und Branche zu achten.

5: Error Prevention Fehlermeldungen sind richtig und wichtig, aber Usability sollte immer anstreben Fehler in Vornherein zu vermeiden.

6: Recognition Rather than Recall Anwendungen sollten eine Wiedererkennungswert haben um einfacher bediehnbar zu sein. Die mentale Last für den Nutzendne um sich an Dinge zu erinnern sollte möglichst gering gehalten werden.

7: Flexibility and Efficiency of Use Erfahrene Nutzende sollen vertraute Aktionen schnell ausführen können.

8: Aesthetic and Minimalist Design Die Gestaltung der Benutzeroberfläche sollte sich nur auf relevante Inhalte konzentrieren.

9: Help Users Recognize, Diagnose, and Recover from Errors Fehlermeldungen sollten in einfacher Sprache formuliert sein (keine Fehlercodes), das Problem genau angeben und konstruktiv eine Lösung vorschlagen.

10: Help and Documentation Nutzende sollten die Anwendung im idealen Fall ohne zusätzliche Erklärung nutzten können. Für seltene Ausnahmen sollte jedoch Hilfestellung bereitgestellt werden.

[Nie94]

2.4 Low-Code vs. Syntaxfrei

Es liegen unterschiedliche Ansätze für Programmierung ohne das tatsächliche Schreiben von Programmcode vor. An dieser Stelle ist es sinnvoll, die Begriffe *Low-Code*, welches jüngst zunehmend an Popularität gewinnt, und *syntaxfreie Programmierung*, wie sie in BlattWerkzeug zu finden ist, voneinander abzugrenzen.

2.4.1 Low-Code

Low-Code als Begriff wurde zuerst von der Marktanalyse-Firma Forrester in den 2010er Jahren geprägt. Low-Code Development Plattforms (LCDP) werden wie folgt definiert:

> Products and/or cloud services for application development that employ visual, declarative techniques instead of programming and are available to customers at low- or nocost in money and training time to begin with [Di +22, S.438].

LCDPs bieten Lösungen zur Entwicklung, Bedienung und Wartung relativ komplexer Softwareanwendungen mit wenig, oder gar keinem Programmcode. Der Aufwand soll dabei möglichst gering gehalten werden. Low-Code-Development-Plattforms sind stark im Bereich der Entwicklung von Business-Applikationen wie Ressourcenplanung, Business Process-Management, Customer-Relationship Management angesiedelt, lassen sich jüngst aber auch in Game Engines wie Godot und Unreal finden. Mit der steigenden Popularität von künstlicher Intelligenz haben in den LCDPs auch KI-Assistenten wie Gemini und Copilot Einzug gehalten. Personen mit technischem Grundverständnis, aber mangelnder Programmiererfahrung sollen mit Hilfe der LCDPs direkt zum Softwareentwicklungsprozess beitragen können. LCDPs finden im Web-Browser und in der Cloud statt, was eine geringe Einstiegshürde, vor allem im Bezug auf Setup und Installation, bietet [Di +22, S.442].

Power Apps von Microsoft Power Apps ist Teil von Microsofts Power Plattform, einer Sammlung von Low-Code-Entwicklungstools. Dort wird cloudbasiert eine schnelle Entwicklungsumgebung bereitgestellt in der benutzerdefinierte Business Anwendungen erstellt werden können. Dabei ist es Benutzenden möglich auf Daten aus verbundenen Systemen, die entweder auf der zugrunde liegenden Datenplattform (Microsoft Dataverse) oder in einer der zahlreichen Online- oder Lokal-Datenquellen (wie SharePoint, Microsoft 365, Dynamics 365, SQL Server usw.) gespeichert sind, zuzugreifen. Laut Microsoft “demokratisiert” Power Apps die Entwicklung von Geschäftsanwendungen, da Nutzende funktionsreiche und benutzerdefinierte Anwendungen erstellen können, ohne Code schreiben zu müssen. Für die Plattform entwickelt und nutzt Microsoft die open source, low-code, general-purpose Programmiersprache Microsoft Power Fx [tap24].

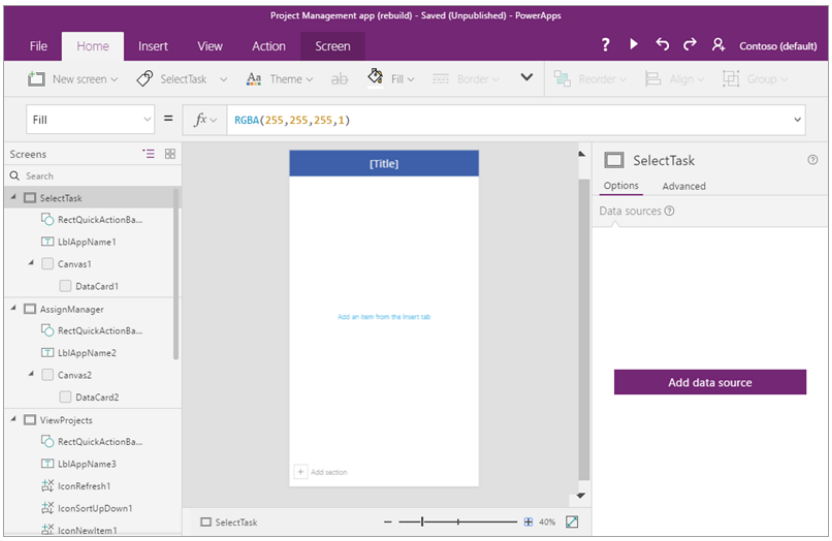

Abbildung 2.2: Ansicht aus dem Editor von Microsoft Power Apps

AppSheet von Google AppSheet ist eine von Google entwickelte No-Code Plattform. Wie auch Power Apps bietet AppSheet Benutzenden die Möglichkeit, umfassende Applikationen und automatisierte Prozesse zu erstellen, ohne auch nur eine einzige Zeile Code schreiben zu müssen.

> The AppSheet Editor helps make creation easier by automatically generating app prototypes and providing smart suggestions for quick customizations. AppSheet also uses spreadsheet-like expressions to incorporate advanced logic to do things like filter data, create dynamic UI elements, and set up workflow automations. [Gooa]

Google kombiniert mit *AppSheet Automation* No-Code und Google KI.

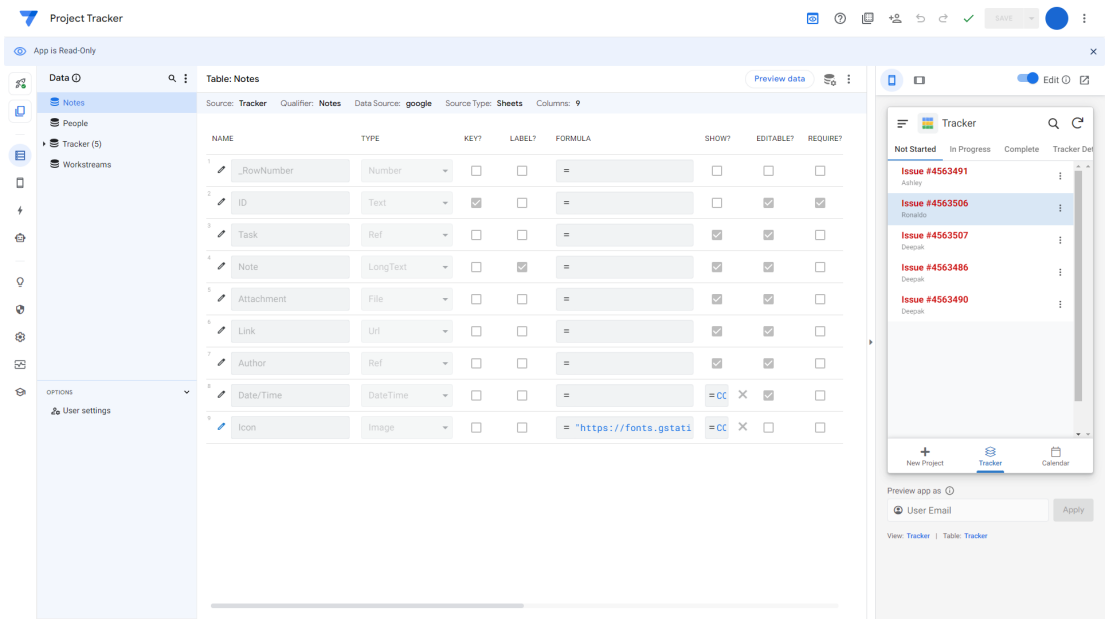

Abbildung 2.3: Ansicht aus dem Editor von Google AppSheet

2.4.2 Syntaxfreie Programmierung

Syntaxfreie Programmiersprachen ermöglichen die Programmierung unabhängig von syntaktischen Details. Bei solchen Programmiersprachen ist beispielsweise das Setzen von Semikola, das Kennzeichen von Blöcken durch geschweifte Klammern oder das Einrücken von Code zur erfolgreichen Kompilierung nicht notwendig. Visuelle Programmiersprachen sind in der Praxis oft syntaxfrei. Bekannte Vertreter sind hierbei *Blockly* und *Scratch*. In diesen Entwicklungsumgebungen ist die Maus das Haupteingabewerkzeug, mit der verschiedene Blöcke, mit vorher bereitgestelltem Programmcode, per Drag & Drop an syntaktisch zulässigen Stellen platziert werden können.

Der Programmcode wird in diesen Entwicklungsumgebungen sozusagen hinter diesen vorgegebenen Blöcken versteckt. Die dadurch entstehende Abstraktion der Programmierkonzepte erschwert Nutzenden jedoch den Umstieg auf “echte” Programmiersprachen. Im Vergleich zu gängigen LCDPs haben Nutzende in syntaxfreien Entwicklungsumgebungen dafür viel mehr Möglichkeit zur kreativen Entfaltung und Entdeckung, da diese oftmals nicht auf den Aspekt der Optimierung von Business-Anwendungen konzentriert sind.

Blockly Blockly ist eine von Google unter der Apache License 2.0 lizenzierte und für Entwickler*innen bereitgestellte Webbibliothek, mit der sich Apps in einem blockbasierten Code-Editor zusammenstellen lassen. Der Editor verwendet Puzzleteile mit Blöcken, um Codekonzepte wie Variablen, logische Ausdrücke, Schleifen usw. darzustellen. Blocky kann dabei auf zwei Arten genutzt werden: Nutzende definieren entweder Puzzleverbindungen und Eingabefelder selbst und Blockly kümmert sich um das Rendern, Verbinden und Platzieren der Blöcke, oder die Nutzenden definieren Strings von Code für die einzelnen Blöcke und Blockly kümmert sich um das Konkatenieren der Strings zu Blöcken [Goob]. So können sich Nutzende auf ihre Zielsetzung konzentrieren. Ein optionaler Blockly Editor ist ebenfalls in BlattWerkzeug implementiert.

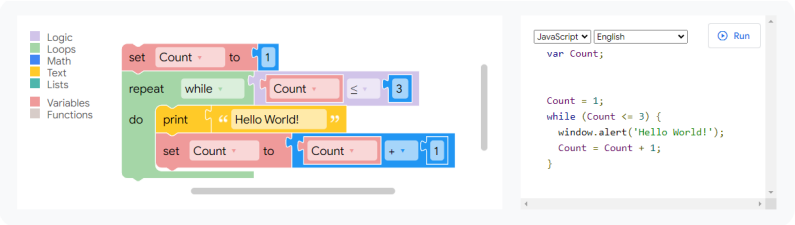

Abbildung 2.4: “Hello World” Programm zusammengesetzt mit Blockly

Scratch

> Scratch ist die weltweit größte Coding-Community für Kinder und eine einfache, visuelle Programmiersprache, mit der junge Menschen ganz einfach digitale Geschichten, Spiele und Animationen erstellen können [Scr].

Der Grundgedanke hinter Scratch ist es, Kinder und Jugendliche früh an Programmierung heranzuführen, um informatorisches Denken, Konzepte zum Lösen von Problemen, Kreativität beim Unterrichten und Lernen, Eigendarstellung und Teamarbeit sowie Chancengleichheit am Computer zu fördern [Scr]. Scratch folgt laut Gründer Mitchel Resnick drei Prinzipien:

> Make it more tinkerable, more meaningful and more social than other programming environments [Res+09, S.63].

Der Prozess, um Programme in Scratch zu erstellen, soll dem Prozess ähneln, der passiert, wenn ein Kind mit Klemmbausteinen spielt. Die Steine können einfach zusammengesteckt werden und die entstehenden Strukturen inspirieren neue Ideen. So besteht Scratches Grammatik aus grafischen “Progammier-Blöcken” die zusammengesetzt werden können, um ein

Programm zu erstellen. Wie bei Klemmbausteinen gibt es an den Blöcken unterschiedlich geformte Konnektoren, die suggerieren, wie Blöcke zusammenpassen. Es gibt keine obskure Syntax oder Zeichensetzung, wie in traditionellen Programmiersprachen [Res+09, S.63]. Wie beim zugrundeliegenden Blockly sind Scratch-Blöcke so geformt, dass diese nur auf Arten zusammenpassen, die syntaktischen Sinn ergeben.

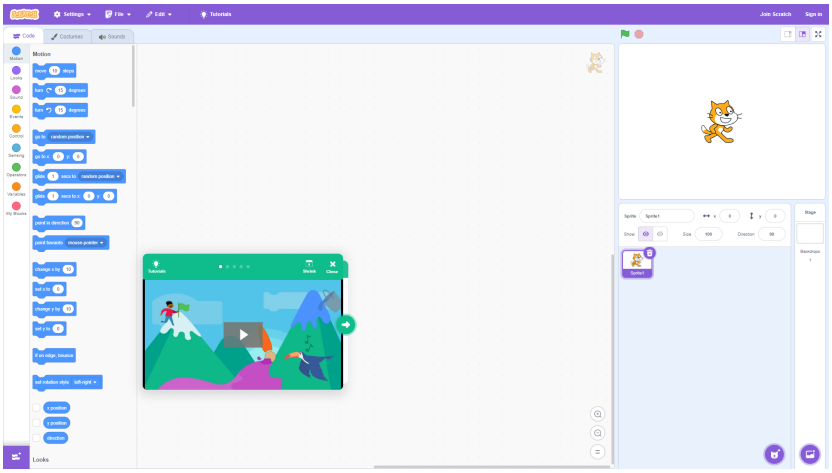

Abbildung 2.5: Ansicht aus dem Scratch Editor

2.5 BlattWerkzeug

Marcus Riemer entwickelte im Rahmen seiner Master-Thesis an der Fachhochschule Wedel 2016 die Lern-Entwicklungsumgebung BlattWerkzeug als Webanwendung für Kinder und Jugendliche. Mit BlattWerkzeug lassen sich, gestützt durch Drag & Drop-Editoren, für bereitgestellte SQLite-Datenbanken Abfragen formulieren und Oberflächen entwickeln. Seit dem Abschluss der Master-Thesis wird BlattWerkzeug im Rahmen eines Promotionsvorhabens und unterschiedlicher Bachelor-Thesen weiterentwickelt.

2.5.1 Was ist ein Block?

Ein Block in BlattWerkzeug entsteht aus dem Zusammenspiel unterschiedlicher Komponenten. Abbildung 2.6 veranschaulicht die zentralen Begriffe und Zusammenhänge mit denen BlattWerkzeug arbeitet.

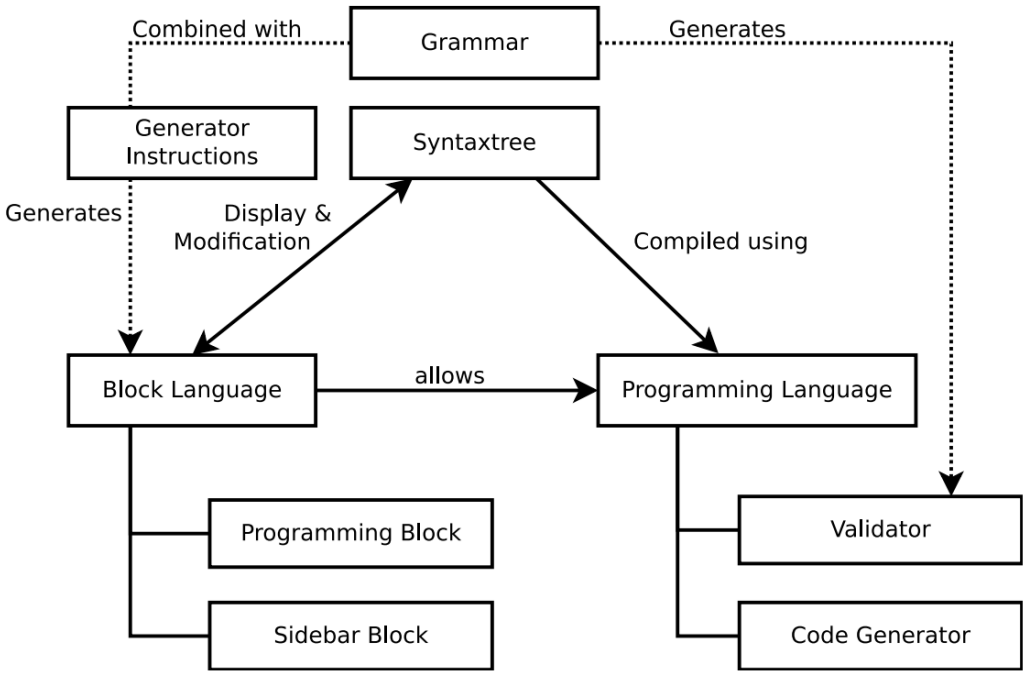

Abbildung 2.6: Zusammenhänge und zentrale Begrifflichkeiten in BlattWerkzeug

Bei der Bearbeitung eines Programms mit einem syntaxfreien Block-Editor sind im Wesentlichen diese vier grundlegenden Datentstrukturen involviert:

Grammar Die Grammatik definiert die grundsätzlich zulässigen Strukturen eines abstrakten Syntaxbaums. Aus dieser Beschreibung lassen sich unmittelbar Validatoren zur Überprüfung von Syntaxbäumen erzeugen. Nimmt man zu dieser Beschreibung noch spezielle *Generator-Anweisungen* (Generator Instructions) dazu, können aus einer Grammatik auch Blocksprachen erzeugt werden.

Syntaxtree, AST Der abstrakte Syntaxbaum repräsentiert die Struktur eines Quelltextes, der mit einem Blockeditor bearbeitet wird. In einer konventionellen Entwicklungsumgebung würde man an dieser Stelle von einer Datei sprechen.

Block Language Die Blocksprache weiß, wie die zu bearbeitende Datei (also ein Syntaxbaum) darzustellen ist. Hier wird definiert, wie die einzelnen Teile des Syntaxbaums visualisiert und editiert werden sollen und welche Blöcke in der Seitenleiste angeboten werden. Außerdem können zu einer Blocksprache auch sprachspezifische Komponenten gehören, zum Beispiel die Anzeige von Abfrageergebnissen bei SQL oder ein Terminal zur Ein- und Ausgabe.

Programming Language Die tatsächliche Kompilierung, Validierung und Ausführung erfolgt anhand einer Programmiersprache. Im Regelfall wird Quelltext für real existierende Zielsprachen wie SQL oder JavaScript erzeugt, welcher dann in einer speziellen Laufzeitumgebung auf dem Server ausgeführt wird [Rie].

2.5.2 SQL Drag & Drop Editor

Über den Drag & Drop Editor findet der Großteil der Interaktion mit dem zu erstellenden Programm statt. Programmcode soll nicht getippt, sondern aus unterschiedlichen Bausteinen mit der Maus zusammengesetzt werden. In einer Seitenleiste sind alle verwendbaren Bausteine aufgelistet. Trotz des syntaxfreien Editors, soll die Anmutung von Programmcode entstehen und auch das Erstellen komplexerer Strukturen möglich gemacht werden. Die Bausteine des Drag & Drop-Editors und deren Kombinationsmöglichkeiten werden durch die zugrundeliegende Grammatik festgelegt. Das Ergebnis des im Drag & Drop-Editors zusammengestellten Codes ist ein Syntaxbaum, der alle “programmierten” Operationen enthält und vom Programm weiterverarbeitet werden kann.

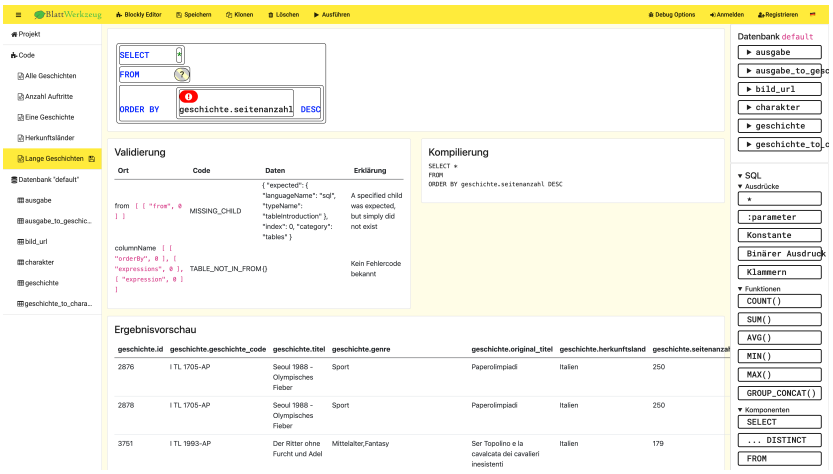

Abbildung 2.7: Screenshot des Drag & Drop-Editors in BlattWerkzeug

Abbildung 2.7 zeigt ein Bildschirmfoto des Drag & Drop-Editors, in dem eine SQL-Abfrage zusammengestellt wird. Im Block-Editor wird die SQL-Abfrage mit Elementen aus der Seitenleiste (rechts) zusammengestellt. Zusätzlich zu Bereichen mit Validierungs- und Kompilierungsausgaben des erstellten AST, wird außerdem eine Ergebnisvorschau mit dem Resultat der Abfrage angezeigt. Eine verbesserte Fehlerdarstellung und Bereitstellung von möglichen Blöcken für Lücken im erstellen Code wird in dieser Arbeit erarbeitet.

2.6 Technische Grundlagen

Diese Arbeit nimmt Anpassungen am Frontend von BlattWerkzeug vor. Zur initialen Entwicklung wurde von Riemer das TypeScript Webframework Angular in Verbindung mit der *RxJS* Bibliothek genutzt.

Laut Riemer bietet sich eine rein clientseitige Visualisierung an, die weitestgehend auf Roundtrips zum Server verzichtet. Außer dem Zugriff auf serverseitige Ressourcen (Datenbank, gespeicherte Ressourcen, gerenderte Seiten) werden alle Operationen im Browser ausgeführt [Rie16, S.94]. Riemer nutzte für die ursprüngliche Entwicklung Angular 2, aktualisierte aber im Rahmen der Weiterentwicklung von BlattWerkzeug einzelne Bereiche auf Techniken aus neueren Versionen des Webframeworks.

2.6.1 Angular

Angular Anwendungen bestehen aus zwei primären Bestandteilen: Komponenten und Services.

Komponenten In Angular ist eine Komponente ein als eigenes HTML-Elemente definiertes Anzeigeelement, welches anhängig von der definierten Anzeige-Logik und den aktuellen Daten den Zustand der Anwendung darstellt. Die unterschiedlichen logischen Bausteine einer Anwendung werden in Komponenten aufgeteilt, welche somit die “building blocks” eben dieser bilden.

Services Sind Daten, Logik und Algorithmen als unabhängig von der Anzeige in der Anwendung definiert, so spricht man von Services. Ihre Stärke liegt in ihrer Modularität und Wiederverwendbarkeit. Sie sind als Klassen zu verstehen, welche Attribute und Methoden definieren, die nicht nur an eine einzelne Komponente gekoppelt sind, sondern von mehreren Komponenten oder auch anderen Services genutzt werden können.

2.6.2 Nutzen von Observables in RxJS

In BlattWerkzeugs Frontend ist das Observer Pattern implementiert. Dabei wird vor allem mit Observables, der Async Pipe aus Angular und den Operatoren der Javascript Bibliothek *RxJS* gearbeitet.

In diesem Abschnitt wird anhand eines Beispiels die Funktionsweise von *RxJS* im Zusammenhang mit dem Observer Pattern veranschaulicht. Dieser Abschnitt ist keine ausführliche Einführung in das Thema *Observer Pattern* oder das Arbeiten mit *RxJS* und dessen genaue Funktionsweise, sondern dient ausschließlich der Illustration des Wissensstands, welcher zum Verständnis der Implementierung dieser Arbeit vorausgesetzt wird.

Zunächst sollten einige gängige Begrifflichkeiten genauer erklärt werden:

Observer Pattern Um das Observable in *RxJS* zu verstehen, ist es zunächst notwendig, das Observer Pattern zu verstehen. Das Observer Pattern ist ein Verhaltensmuster, welches der Weitergabe von Änderungen an einem Objekt (Observable) an die von diesem Objekt abhängigen Strukturen dient. Diese Weitergabe erfolgt durch eine Subscription auf das Observable.

Der Datenstrom Eine Reihe von über einen bestimmten Zeitraum gesammelten Datenwerten, werden als Datenstrom dargestellt. Diese Sammlung kann etwas Simples sein, wie ein einfaches Inkrement von Zahlen oder etwas Komplexeres, wie die Datenwerte von Eingaben in Formulare, die über WebSockets oder API-Antworten übermittelt werden.

Observables Ein Observable ist eine Funktion, die im Laufe der Zeit einen Datenstrom an einen Observer weitergeben kann. Dies kann sowohl synchron, als auch asynchron geschehen.

Observer Ein Observer ist ein Objekt, das ein Observable beobachtet, um benachrichtigt zu werden, wenn dieses mit dem Datenstrom interagiert. Ein Observer hat drei Methoden: `next()`, `error()` und `complete()`. Diese Methoden werden jeweils durch das Observable aufgerufen, wenn neue Werte über den Datenstrom gesendet werden, wenn während des Sendevorgangs ein Fehler auftritt oder wenn das Observable abgeschlossen ist und keine weiteren Werte mehr sendet. Observer können verwendet werden, um die von einem Observable ausgegebenen Werte zu konsumieren und Folge-Prozesse auszulösen, wie z. B. die Aktualisierung einer Benutzeroberfläche oder die Durchführung von Berechnungen.

Operatoren Um den Datenstrom, der vom Observable ausgeht zu manipulieren, existieren eine Vielzahl von Funktionen, die Operatoren genannt werden. Jeder Operator nimmt

einen oder mehrere beobachtbare Datenströme als Eingabe entgegen und gibt einen neuen beobachtbaren Datenstrom als Ausgabe zurück. Operatoren können miteinander verkettet werden, um komplexe Datentransformationspipelines zu bilden, die es Ihnen ermöglichen, den Datenstrom zu manipulieren und zu transformieren. Dies geschieht bei *RxJS* innerhalb der `.pipe()` Methode.

Async pipe Die Async pipe ist ein Feature aus Angular, dass den Prozess des Subscribens und Unsubscribens auf Obeservables im Template vereinfacht. Es regelt automatisch den Subscription Lifecycle (Abb. 2.8) und aktualisiert das Template mit den Werten aus dem Datenstrom.

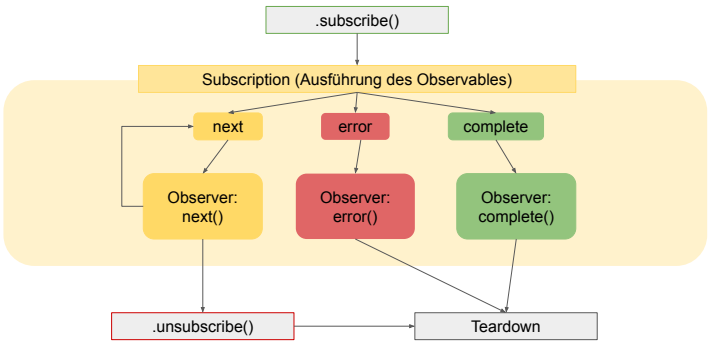

Abbildung 2.8: Darstellung des Subsciption Lifecycle

Beispiel zur Nutzung von *RxJS*

In diesem Beispiel wird ein Name nach Eingabe in ein Textfeld darauf geprüft, ob dieser häufiger von männlichen oder weiblichen Personen genutzt wird. Bei der Eingabe werden automatisch Anfragen an die Gender-API geschickt, deren Antwort dann unter dem Eingabefeld angezeigt wird (Abb. 2.10). Vor der Eingabe wird “null” angezeigt (Abb. 2.9).

Abbildung 2.9: Beispielanwendung ohne Eingabe

Abbildung 2.10: Beispielanwendung nach Eingabe

```
<input type="text" #txtField>
<pre>{{ observable$ | async | json }}</pre>
```

Listing 2.1: HTML Template des Beispiels

```
@Component({
    //...
    templateUrl: './app.component.html',
    styleUrl: './app.component.scss'
})

export class AppComponent implements AfterViewInit {
    readonly genderGuess = inject(GenderGuessService)
    @ViewChild('txtField', {static:true}) textField!:ElementRef;
    observable$ : Observable<any> = of(null)

```

```
12      ngAfterViewInit(): void {
13          this.observable$ = fromEvent<any>
14                                  (this.textField.nativeElement,'input')
15          .pipe(
16              debounceTime(1000),
17              map((event:any) => event.target.value),
18              distinctUntilChanged(),
19              switchMap((s) => this.genderGuess.getData(s)),
20          )
21      }
22  }
```

Listing 2.2: app.component.ts des Beispiels

In Zeile 8 wird der `GenderGuessService`, welcher die Kommunikation mit der Gender-API, übernimmt in die Komponente injiziert. Angular unterstützt den Ansatz von Service und Dependency Injection um sicher zu stellen, dass in Komponenten wirklich nur die direkte Interaktion mit den Nutzenden stattfindet.

Es folgt in Zeile 9 eine View Query für das Textfeld, in welches der Name eingegeben werden kann. Es wird der Selector `'txtField'` angegeben, der sich im input tag des zur Komponente gehörigen HTML Template befindet.

In Zeile 10 wird dann das Observable definiert und mit dem Wert `of(null)` initialisiert. Der Name des Observables wird dabei mit einem $ versehen, damit es bei der weiteren Nutzung von einer Variable eines regulären Datentyps unterscheidbar bleibt. Die Zuweisung des initialen Wertes erfolgt durch den Gebrauch des `off()` Operators.

In Zeile 12- 19 des Beispielcodes wird definiert, was im `ngAfterViewInit()` des component lifecycles stattfinden soll. Zunächst wird festgelegt, dass die kommenden Operationen durchgeführt werden sollen, wenn es zu einer Interaktion mit dem `'input'` Feld kommt. Danach wird die `pipe()` Funktion von *RxJS* genutzt um die darauffolgenden Operatoren miteinander zu verketten.

Es sollen möglichst wenige HTTP Requests an die API gesendet werden, weshalb im Beispiel der `debounceTime()` Operator verwendet wird. Dieser fundiert als eine Art Verzögerung, die sich nur die letzte Änderung am Observable merkt und nur diesen Wert nach der angegebenen Zeit weitergibt.

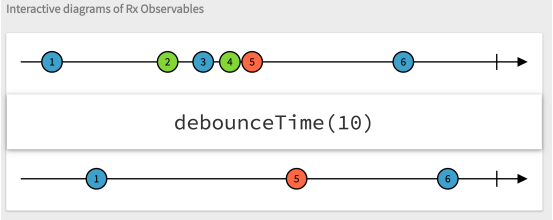

Abbildung 2.11: [RxM] Darstellung des debounceTime() Operators

Der `map()` Operator nimmt den Wert vom ViewChild event target, also dem input Feld, und merkt sich diesen im `observable$`. Hier wird mit Hilfe des `map()` Operators ein String verarbeitet, der in das Feld geschrieben wurde.

Der Operator `distinctUntilChanged()` sorgt im Zusammenspiel mit `debounceTime()` dafür, dass nur HTTP Requests gestellt werden, wenn sich der Input nach Ablauf des angegeben Zeitfensters verändert hat.

`switchMap()` verarbeitet das Observable mit dem eingegeben Text und schickt über den `GenderGuessService` die Anfrage an die API. Wie bei `map()` wird hier der ursprüngliche Wert nicht überschrieben, sondern in einer Lambda-Funktion verarbeitet und ausgegeben. Es wird kein einfaches `map()` für diese Operartion verwendet, da das Ergebnis mit der Antwort der API wieder ein Observable sein soll. Diese Antwort wird im HTML template mit einer Kombination aus Angulars Async Pipe und der Json Pipe entpackt und angezeigt.

3 Anforderungsanalyse

Ziel dieser Arbeit ist es, das bereits vorhandene Projekt in Hinsicht auf UI und UX zu überarbeiten und dazu einen Filter für valide Erweiterungen von unvollständigen Syntaxbäumen zu ergänzen. Um die Erfahrung der Nutzenden mit BlattWerkzeug tatsächlich zu verbessern, ist es angebracht, zunächst deren Bedürfnisse und Anforderungen an die Software zu analysieren. Darüber hinaus muss bei der Entwicklung und Umsetzung des Filters und anderer Änderungen Rücksicht auf die Implementierung und technischen Grundlagen des bereits bestehenden Projektes genommen werden.

3.1 Vorhandenes Projekt

Die von Marcus Riemer im Rahmen seine Master-Thesis an der Fachhochschule Wedel erstellte Lehr-und Entwicklungsumgebung BlattWerkzeug richtet sich in erster Linie an Kinder-und Jugendlich im Mittelstufenalter. Vorrangig sollen praktische Kenntnisse zur Abfrage, Manipulation und Visualisierung von komplexen Datenbeständen in Anlehnung an die Projektideen von Lehrplänen und Fachanforderungen für das Fach Informatik vermittelt werden [Rie16, S.2]. Im Rahmen seines Promotionsverfahrens entwickelt Riemer BlattWerkzeug stetig selbst weiter.

3.1.1 Drag and Drop Editor

Dieser Abschnitt stellt zunächst genauer die bereits vorhandene UI des Editors in BlattWerkzeug vor.

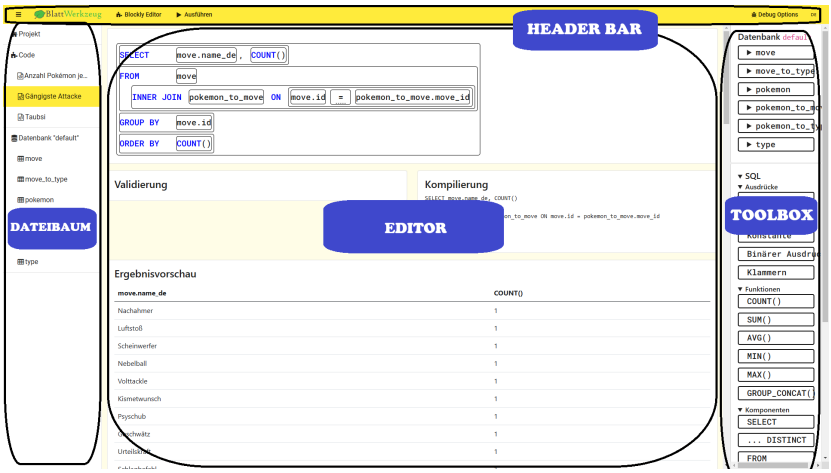

Abbildung 3.1: Ansicht des Drag & Drop Editors in BlattWerkzeug mit Kennzeichnung der unterschiedlichen Bereiche

Die visuelle Oberfläche des Editor in BlattWerkzeug ist der einer typischen Entwicklungsumgebung nachempfunden. So spiegelt sich im Design eine klassische Dreiteilung in Dateibaum, Editor und Toolbox wieder.

Dateibaum

Der Bereich für den Dateibaum befindet sich auf dem linken Teil des Bildschirms und bietet eine Übersicht zu allen Projektdateien. Der gesamte Bereich kann je nach Belieben durch ein Burgermenü versteckt werden. So können Nutzende sich besser auf die Arbeit im Editor fokussieren und erhalten bei Bedarf auch etwas mehr Platz im Editor, wenn Syntaxbäume beispielsweise sehr komplex werden und damit auch mehr Raum in Editor einnehmen.

Code Der Punkt *Code* im Dateibaum dient zum Anlegen einer neuen Code Ressource in der eine neue Query erstellt werden kann. Eine Code Ressource ist mit einer leeren Datei zu vergleichen, in der Benutzende von Null aus Blöcke zu einem AST zusammensetzen können. Die verfügbaren Blöcke ergeben sich aus der mit dem Projekt verknüpften Datenbank und der für die Code Ressource angegebenen Block Language.

Für die Entwicklung des Filters entscheidend ist hier, dass immer eine Code Ressource geladen wird, wenn im Editor etwas bearbeitet wird. So ist es sinnvoll, den Filter für die möglichen Blöcke in den Service, der sich um die Code Ressource kümmert, einzubetten.

Datenbank Der Punkt *Datenbank* zeigt das Datenbankmodell an, welches in dem ausgewählten Projekt verwendet wird. Das Modell wird in Form eines Datenbankdiagramms dargestellt. Zusätzlich sind alle Tabellen der Datenbank separat einsehbar. Benutzende können die Struktur der Datenbank jederzeit über einen integrierten Datenbankeditor bearbeiten.

Editor

Der Editor ist der Bereich, in dem Blöcke zu einem AST zusammengesetzt werden und automatisch ausgeführt werden können. Nach einer Änderung im zusammengesetzten Baum wird dieser automatisch neu validiert und ausgeführt.

Validierung Tritt ein Fehler bei der Validierung des Baumes auf, wird dieser im Feld "Validierung" aufgeführt. Als Hilfestellung zum Lösen des Fehlers werden der Ort im AST, an dem der Fehler auftritt, der dem Fehler entsprechende Fehlercode, und die Daten, die an der entsprechenden Fehlerstelle zu finden sind, ausgegeben. Diese Ausgabe hilft Nutzenden insofern, dass eingesehen werden kann, welche Daten das System beispielsweise erwartet.

Kompilierung Neben dem Feld für die Validierung, gibt es im Editor ein Feld in dem Nutzende den von ihnen erstellten Code so sehen können, wie er aussehen würde, wenn er in einem konventionellen Texteditor geschrieben wäre. Dieser dient aktuell eher als Debug Bereich für Entwickelnde, hat aber Potenzial dafür genutzt zu werden, Schüler*innen für die tatsächliche Darstellung von Code zu sensibilisieren.

Ergebnisvorschau Am unteren Ende des Editors befindet sich ein Feld, in dem das Ergebnis des erstellten AST angezeigt wird. Bei SQL-Datenbankabfragen ist das beispielsweise eine Tabelle mit den abgefragten Daten. Diese Ergebnisvorschau kann auch leer sein, wenn der Baum nicht korrekt validiert werden konnte.

Toolbox

Die Toolbox ist in BlattWerkzeugs Fall die Seitenleiste, in der sich alle Blöcke befinden, die zu AST zusammengesetzt werden können. In der Seitenleiste wird zwischen Datenbank Blöcken und Sprach Blöcken unterschieden. Die Datenbank-Blöcke werden automatisch, aus der mit

dem Projekt verknüpften Datenbank, generiert. Es entstehen immer jeweils ein Block für die gesamte Tabelle, ergänzt durch Blöcke der einzelnen enthaltenen Spalten. Sprachblöcke hingegen ergeben sich aus der, für die Code Resource festgelegten und in Grammatiken definierte, Blocksprache. Für SQL sind diese Blöcke aufgeteilt in Ausdrücke, Funktionen und Komponenten.

Header Bar

In der Header Bar befinden sich zusätzliche, nicht projektspezifische Optionen zur Bearbeitung des Projekts. Hier finden Nutzende beispielsweise Shortcuts zum Anlegen neuer Tabellen oder Importieren neuer Daten. Der AST kann über die Funktionen im Header manuell ausgeführt werden. Außerdem ist es an dieser Stelle möglich Code Resourcen zu klonen, speichern und löschen. Zusätzlich kann im Header zwischen dem BlattWerkeugeigenen Editor und dem integrierten Blockly Editor umgeschaltet werden.

3.1.2 Zielgruppe

Bei der Zielgruppe von BlattWerkzeug handelt es sich hauptsächlich um Schüler*innen im Alter von 13 Jahren bis 17 Jahren. Die Software ist darauf ausgelegt, unter Aufsicht einer Lehrperson genutzt zu werden und setzt ein geringes Maß an Grundkenntnissen in EDV voraus. [Rie16, S.23]

BlattWerkzeug sieht folgendes Anforderungsprofil für die erfolgreiche Nutzung vor:

Grundlegende PC-Anwendung BlattWerkzeug setzt grundlegende Kenntnisse zur Anwendung von Computern voraus, darunter Klarheit über Begriffe wie “Speichern” und “Drag & Drop”, sowie Vertrautheit in der Bedienung eines Internetbrowsers. Tiefergehende Kenntnisse über die Funktionsweise von Computern und deren Dateisystemen sind hingegen nicht notwendig.

Tabellenkalkulation Grundkenntnisse im Umgang mit Tabellenkalkulationsprogrammen werden als eine sinnvolle Vorstufe zur Arbeit mit BlattWerkzeug gesehen. So können Nutzende die Strukturierungsmöglichkeiten von Datenbeständen, insbesondere in SQL-Projekten, besser verstehen.

Englischkenntnisse Wie viele andere Programmiersprachen auch, wurden SQL und HTML nicht ins Deutsche übersetzt. Um die Funktionsweise dieser Sprachen zu verstehen, ist es daher wichtig, die Begriffe, die den Befehlen zugrunde liegen, zu kennen.

Personas

Personas sind ein gängiges Hilfsmittel, um ein tieferes Verständnis für Ziele und Motivationen der Zielgruppe zu erlangen und gleichzeitig die Anforderungen, an das System präzise darzustellen. Die für BlattWerkzeugs Zielgruppe erstellten Personas werden im folgenden vorgestellt.

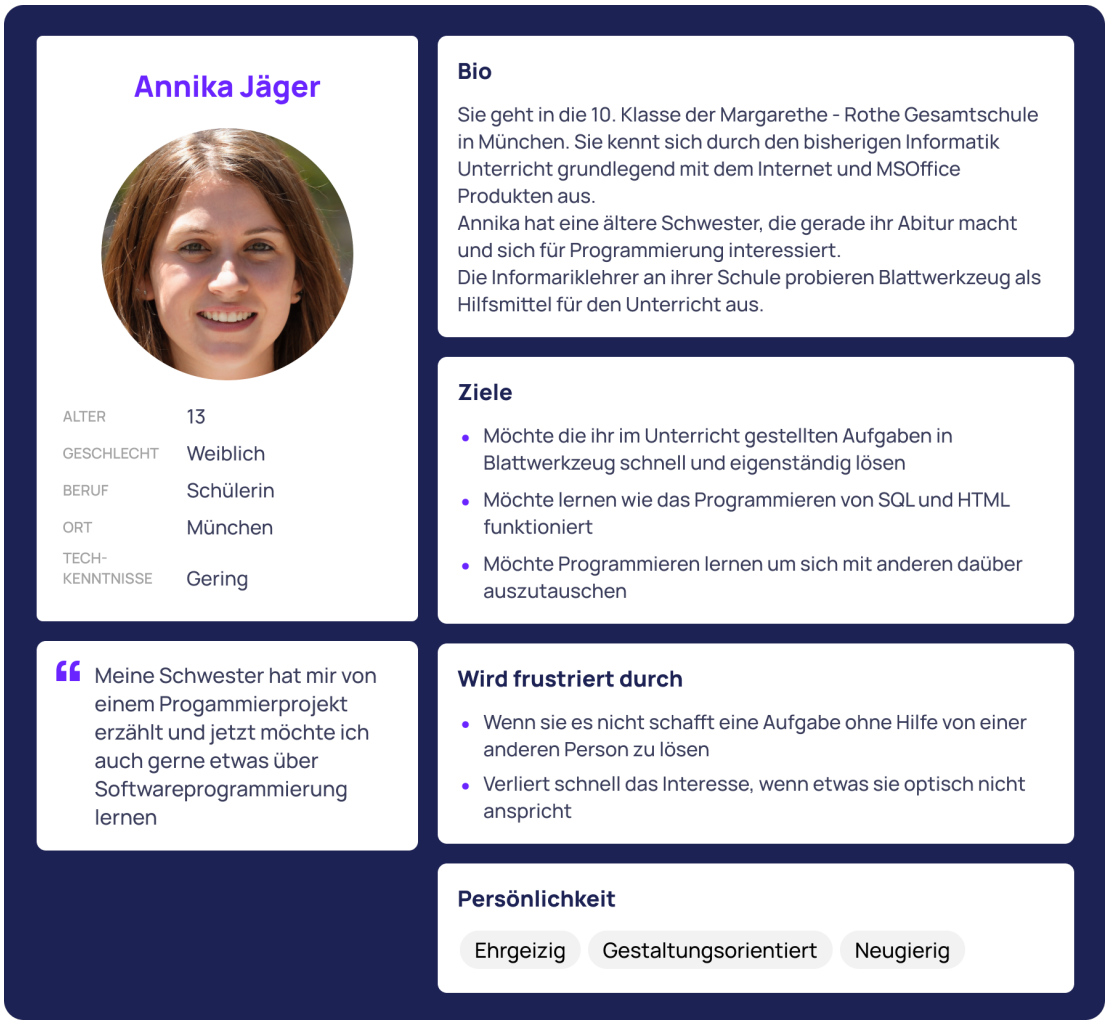

Abbildung 3.2: Persona Neuling Annika Jäger

Abbildung 3.3: Persona Erfahrener User Yusef Bahir

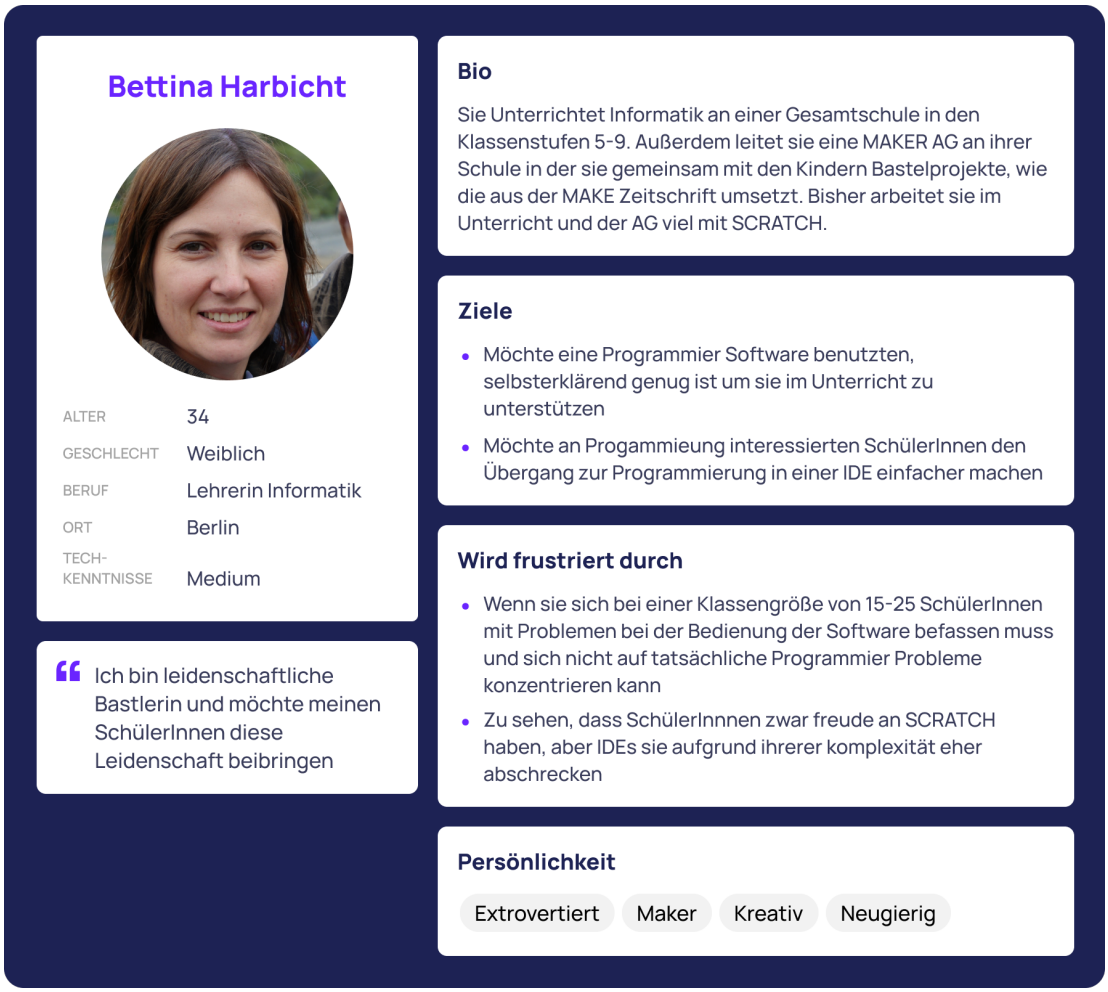

Abbildung 3.4: Persona Lehrperson Bettina Harbicht

Basierend auf der Zielgruppendefinition und dem dazugehörigen Anforderungsprofil gehen drei relevante Personas hervor: Ein Neuling, in Form von Annika Jäger (Abb.3.2), ein erfahrener Nutzender in Form von Yusef Bahir (Abb.3.3) und eine Lehrperson in Form von Bettina Harbicht (Abb.3.4). Es handelt sich hierbei zwar um Personen mit unterschiedlichem Grad an Technikkenntnissen, keine*r von ihnen ist jedoch professionelle*r Informatiker*in. Alle haben ein Interesse daran, in BlattWerkzeug einen barrierefreien Zugang zur Programmierung zu finden. Wichtig ist ihnen, dass selbstständig in BlattWerkzeug gearbeitet werden kann, ohne, dass Fehler zu Blockaden führen, die nicht selbstständig und effizient gelöst werden können.

3.2 Anforderungen

Nach der Prüfung von BlattWerkzeug auf Schwachstellen wurden unter Rücksichtnahme der Zielgruppenanalyse folgenden Anforderung zur Verbesserung der UX durch die Anpassung der UI und das Einfügen eines Filters für unvollständige AST erarbeitet.

3.2.1 Nicht-Funktionale Anforderungen

Für eine gute User Experience sind nicht-funktionale Anforderungen genauso so wichtig wie die funktionalen Anforderungen. Sie geben an, in welcher Qualität, die Anforderungen im System umgesetzt werden [Mos12, S.102]. Da diese Arbeit die Verbesserung der User Experience einer bereits bestehenden, funktionierenden Software anstrebt, werden mehr nicht-funktionale Anforderungen gestellt, als funktionale.

NA-1 Lesbarkeit Die einzelnen Blöcke in der Sidebar sollen überarbeitet werden. Texte sollen nicht mehr über den Rand hinauslaufen, sondern am Ende des Blocks ausgepunktet werden. Die Größe der Blöcke in der Sidebar soll dabei unverändert bleiben. Beim Hovern mit der Maus über die Blöcke soll eine Scroll-Animation ausgeführt werden, damit die komplette Beschriftung des Blocks gelesen werden kann.

NA-2 Übersicht Die Blöcke im Editor selbst sollen überarbeitet werden. Hierbei müssen die Abstände zwischen Text und Umrandung angepasst werden, damit die Blöcke nicht mehr so gedrungen wirken.

NA-3 Verständlichkeit Die Validierungsanzeige soll verbessert werden, damit dort angezeigte Fehlermeldungen auch von unerfahren Programmierenden verstanden werden können und diese eine echte Hilfestellung zur Lösung von Fehlern im Baum darstellt. Dies soll durch das Hinzufügen einer Spalte mit einer menschenlesbaren Erklärung des Fehlercodes erreicht werden.

3.2.2 Funktionale Anforderungen

Funktionale Anforderungen definieren die konkreten Aktionen, die eine Anwendung nach der Eingabe durch Benutzende ausführen soll oder nicht.[Mos12, S.90]

FA-1 Blockfilter Die Blöcke in der Seitenleiste sollten gefiltert werden, sodass nur valide Blöcke für das aktuell vom Nutzenden ausgewählte Loch im AST angezeigt werden. Nicht

valide Blöcke aus der Seitenleiste sollen beim Klick auf die Lücke im Syntaxtree automatisch ausgeblendet werden, sodass Nutzende gar nicht die Möglichkeit bekommen, potenziell falsche Blöcke auszuwählen. Der Ablauf des Filterns ist im abgebildeten Sequenzdiagramm 3.5 zu sehen.

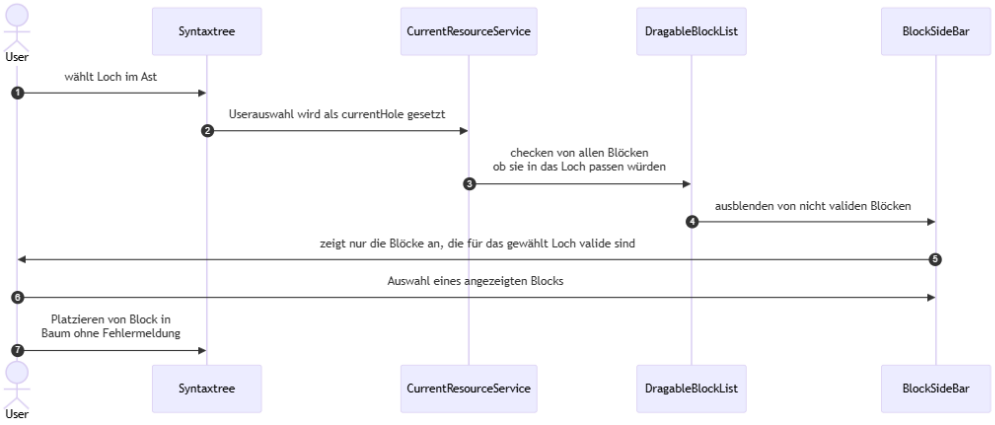

Abbildung 3.5: Sequenzdiagramm für das Auswählen und Setzen eines validen Blocks in den AST

FA-2 Lückenauswahl Die zu bearbeitende Lücke im AST soll anklickbar sein, damit Nutzenden klarer, ist an welcher Stelle der AST gerade bearbeitet wird. Ein zusätzliches Highlighting des Loches unterstützt Nutzende dabei, zu erkennen, welche Stelle im AST gerade bearbeitet wird. Es ist wichtig, dass Nutzende sich im Klaren darüber sind, für welche Stelle im Baum die Blöcke gerade gefiltert werden.

3.2.3 Berücksichtigung der UX Regeln

In diesem Abschnitt wird darauf eingegangen, inwiefern die in Kapitel 2 benannten Laws of UX bei den vorgeschlagenen Änderungen Anwendung finden.

Tesler's Law Dadurch, dass die potenziellen Blöcke vorgefiltert werden, wird auch die Komplexität der zu lösenden Aufgabestellungen verringert. Es können keine nicht zulässigen Blöcke mehr eingesetzt werden, da sie nicht mehr angezeigt werden. Die Anzeige und die Interaktionsmöglichkeiten für Nutzende werden also nur noch auf sinnvolle Handlungen beschränkt.

Jakob's Law Die allgemeine Gestaltung des dreigeteilten Editors von BlattWerkzeug folgt bereits Jakob's Law, da diese Gestaltung gängig für Softwareeditoren, aber auch im allgemeinen Umgang mit Desktopanwendungen ist. Familiarität wird beachtet durch das Highlighting einer angeklickten Fläche und die Scrollbarkeit von Elementen, die zu groß sind.

Hick's Law Ähnlich wie bei Tesler's Law gilt auch hier, dass die Komplexität der zu lösenden Aufgaben durch ein Vorfiltern der möglichen Blöcke verringert wird. Dadurch können sich Nutzende, wie in Hick's Law vorgesehen, schneller durch die Anwendung bewegen. Überflüssige, nicht erlaubte Auswahlmöglichkeiten werden ausgeblendet, wodurch das Erreichen des angestrebten Ergebnisses schneller zu bewältigen ist.

Peak-End-Rule Beim Zusammensetzen des AST aus den vorgegebenen Blöcken befinden sich Nutzende üblicherweise auf dem Peak der Nutzung von BlattWerkzeug, da dort die ihnen vorschwebende Funktion des AST in die Realität umgesetzt werden muss. Dort befindet sich während der User Experience ein kritischer Punkt mit viel Frustrationspotenzial, wenn Nutzende glauben ein Block wäre geeignet, sie diesen beginnen zu ziehen und dann aber feststellen, dass der Block gar nicht erlaubt ist an der Stelle wo er eingesetzt werden soll. Durch das Ausblenden dieser nicht erlaubten Blöcke wird die Erfahrung auf dem Peak positiver.

Aesthetic-Usability Effect Die allgemeine Überarbeitung von überlaufendem, zu eng angeordnetem Text macht einen ästhetisch ansprechenderen Eindruck auf Nutzende.

3.2.4 Geplante Funktionsweise des Filters

Der Plan ist es, den Filter für die Blöcke in den bereits vorhandenen *current-coderesource.service* zu integrieren.

Grundsätzlich wird der Filter folgende Parameter benötigen:

1) Den aktuellen AST in Bearbeitung

2) Den Validator, welcher den AST auf Richtigkeit überprüft

3) Die Position im AST, an der sich die aktuell zu bearbeitende Lücke befindet

4) Den Block aus der Seitenleiste, der in den AST eingesetzt werden soll

Beim Filtern soll der übergebene Block an der Position der Lücke in den aktuellen AST eingesetzt werden. So entsteht ein potenzieller AST, der anschließend vom Validator auf seine Richtigkeit überprüft wird. Diese Prüfung soll nur für die ausgewählte Lücke erfolgen. Die

Umsetzung des Filters, speziell das Einsetzen der in der Seitenleiste verfügbaren Blöcke, soll über die Verwendung von Observables gelöst werden. Um nicht erwünschte Blöcke auszublenden, sollte es möglich sein, die selbe Logik auszunutzen, die bereits die Lücken beim Drag & Drop rot oder grün einfärbt.

4 Implementierung

Dieser Implementierungsabschnitt gliedert sich in die Beschreibung der unterschiedlichen Iterationen des Filters während des Entwicklungsvorgangs. Beginnend mit der Durchführung von Specification Tests, gefolgt von der initialen Umsetzung des Filters, in der für jeden Block immer eine Validierung des gesamten AST stattfindet, über das schrittweise Auslagern der Filterparameter, bis hin zur Nutzung von Observables und der `AllowsChildType()` Funktion zur Optimierung der Performance bei der Validierung der eingesetzten Blöcke.

4.1 Vorbereitung

Die entwickelte Filterfunktion wird hauptsächlich zum *current-coderesource.service.ts* hinzugefügt, da dieser Service die einzelnen Code Resourcen repräsentiert, die vom Nutzenden bearbeitet wird. Der Service stellt eine Verbindung zwischen der tatsächlichen Resource und Komponenten wie dem Validator her. Diesem Service sind bereits sowohl der aktuelle Syntaxtree, als auch die Blocksprache, die in der Code Resource verwendet werden, bekannt. Darüber hinaus ist die Blocksprache auch notwendig, um den Validator zu verwenden.

4.1.1 Speicherung des ausgewählten Loches

Nachdem ein Ort für die Implementierung des Filters gefunden wurde, musste zunächst eine neue Funktion erstellt werden, die das vom Nutzenden zu bearbeitende Loch zwischenspeichert. Dazu wurde ein `onMouseClick()` event in der *block-render-drop-target.html* hinzugefügt und dann im *current-coderesource.service.ts* als Observable `currentHoleLocation$` gespeichert.

Zusätzlich wurde ein farbiger Schatten um das ausgewählte Loch herum hinzugefügt, wenn dieses angeklickt wird. So sollen Nutzende immer erkennen können, welches Loch gerade bearbeitet wird (Abb. 4.1).

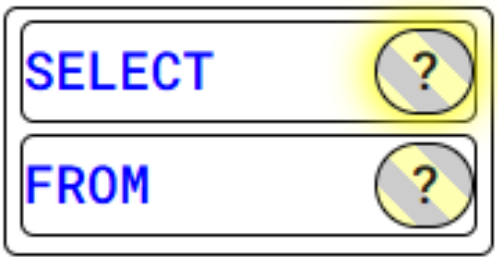

Abbildung 4.1: Ausgewähltes Loch mit Highlight und nicht ausgewähltes Loch ohne Highlight

4.1.2 Umgang mit mehreren Löchern im AST

Es musste überlegt werden, wie damit umgegangen werden soll, wenn ein AST mehrere Löcher beinhaltet. Das ist insofern ein Problem, als dass die Validierung und Filterung der Blöcke nur auf das vom Nutzenden aktuell angeklickte Loch bezogen werden soll. Es ergaben sich zwei Möglichkeiten für den Umgang mit diesem Problem bei der Entwicklung des Filters:

1. **Ausblenden:** Alle Fehler, die im AST vor dem Einsetzen eines potenziellen Blocks auftreten, werden zwischengespeichert. Nach Erstellung und Validierung des potenziellen AST werden die dann aufgetretenen Fehler mit den vorherigen verglichen. Weist dieser Vergleich keine Unterschiede auf, so ist der potenzielle AST valide und der Block kann angezeigt werden.

2. **Fokus:** Bei der Validierung des einzufügenden Blocks wird nur die Node, in der sich das ausgewählte Loch befindet und die jeweilige Parent Node auf Fehler geprüft. Children Nodes der Lücke oder andere Fehler irgendwo im AST interessieren an diesem Punkt beispielsweise gar nicht und werden ignoriert.

Für die Umsetzung des Filters wurde zunächst die Fokus-Möglichkeit verfolgt.

4.1.3 Specification Testing

Um vor der Entwicklung greifbarer zu machen, wie das Einsetzten von Blöcken in einen unvollständigen AST Fehler verursachen kann, wurde eine Reihe von Specification-Tests erstellt und durchgeführt. In technischer Hinsicht werden diese Tests mit Hilfe der Jasmine-Bibliothek durchgeführt. Das Verhalten der zu implementierenden `currentHoleMatchesBlock()` Funktion wird durch eine Verkettung von `.expect().toEqual()` Funktionen geprüft. Die Bibliothek stellt neben `.toEqual()` auch noch andere Vergleiche wie `isUndefined()`

oder `exists()` zur Verfügung. Wird innerhalb eines Testfalls auch nur eine einzige dieser Prüfungen nicht wie erwartet ausgewertet, so wird der Testfall als insgesamt fehlgeschlagen markiert und aufgelistet. Es wurden Testdaten für unterschiedlich komplexe Bäume mit unterschiedlichen Anzahlen an Lücken angelegt, um möglichst viele mögliche Szenarien zu testen (Abb. 4.2).

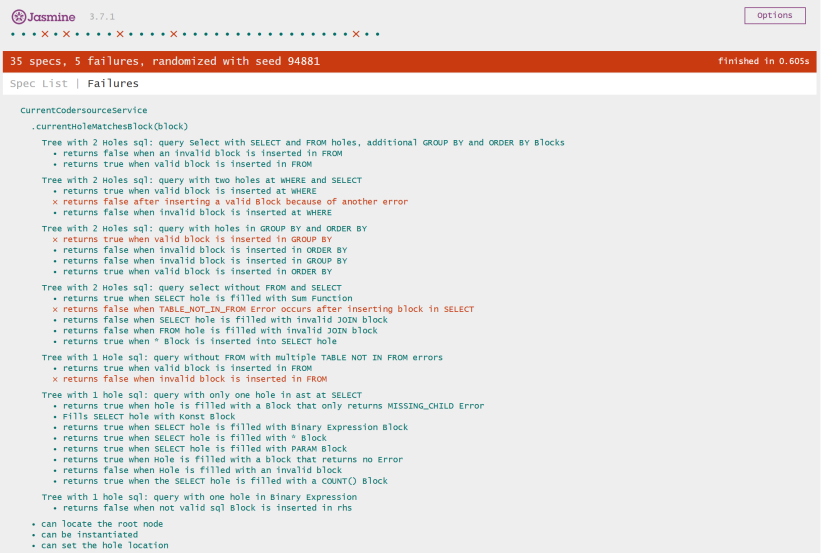

Abbildung 4.2: Jasime Testrun

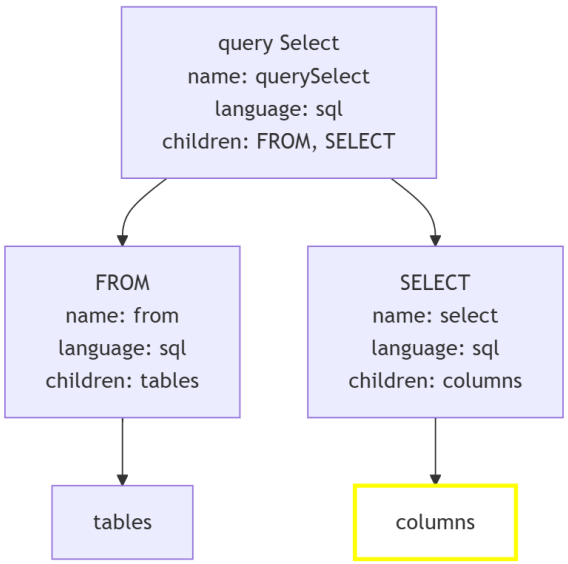

Abbildung 4.3: Baumdarstellung der CurrentHoleLocation im AST

Funktionsweise der Tests

Jedes `describe()` im Test beschreibt einen AST mit unterschiedlichen Blöcken und unterschiedlicher Anzahl und Orten von Lücken. Das genaue Aussehen des AST wird zu Beginn eines `describe()` Blocks als eine Konstante gespeichert. Anschließend wird eine Code Resource mit dem gegebenen Baum instanziiert und ein Block beschrieben, der in diesen Baum eingefügt werden soll. Damit die `currentHoleMatchesBlock()` Funktion im Test ausgeführt werden kann, muss noch eine `currentHoleLocation$` gesetzt werden. Das Löcher im Baum immer daran zu erkennen sind, dass leere Children vorhanden sind, ist in (Abb.4.1) und (Abb. 4.3) zu sehen. Zum Schluss wird der zuvor definierte Block in den vorgegebenen Syntaxbaum eingesetzt und es wird eine Assertion darüber getroffen, ob das Ergebnis der Validierung true oder false sein soll. Der folgende Code zeigt, wie ein Test für den in Abbildung 4.1 gezeigten AST aussehen kann.

```
describe(‘Tree with 2 Holes sql:
  query select without FROM and SELECT‘, () => {
    const treeDescTwoHolesSelectFrom = {
        name: "querySelect",
        language: "sql",
        children: {
          from: [{name: "from", language: "sql",}],
```

```
8           where: [],
9           select: [{name: "select", language: "sql",}],
10          groupBy: []
11        },
12    };
13
14    it(‘returns true when * Block is inserted into SELECT hole‘,
15       async () => {
16        const s = await instantiate(treeDescTwoHolesSelectFrom);
17
18        s.setCurrentHoleLocation([
19          ["select", 0], ["columns", 0],
20        ]);
21
22        const block: NodeDescription = {
23          name: "starOperator", language: "sql",
24        };
25
26        const result = await s.currentHoleMatchesBlock(block);
27        expect(result).toBeTrue();
28      }
29    );
30 }
```

Listing 4.1: Test aus current-coderesource.service.spec.ts

Mögliche Fehler, die beim Einfügen von Blöcken auftreten können

Die folgenden Fehler konnten als mögliche Problemfälle und damit als relevant für den Filter identifiziert werden.

MISSING CHILD ERROR Dieser Fehler tritt auf, wenn ein Block einen weiteren Block fordert (Abb. 4.4). Der Fehler wird bereits in der ersten Iteration des Filters in der Validierung nicht beachtet, da es nicht relevant ist, ob nach dem Einfügen des Blocks noch ein weiterer Block eingefügt werden muss.

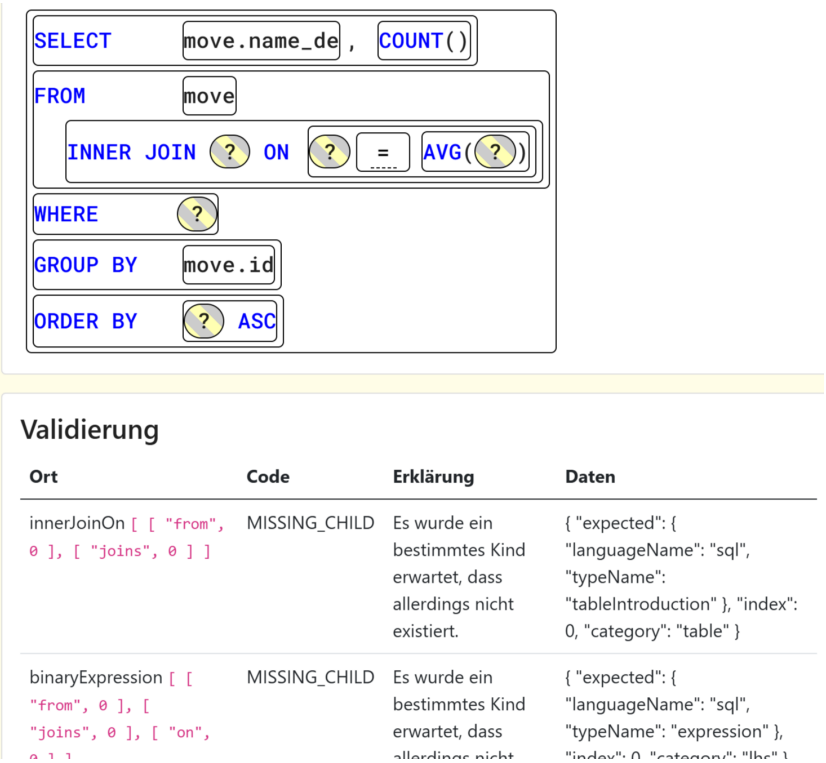

Abbildung 4.4: AST mit erlaubten Blöcken, die Missing Child Error auslösen, darunter INNER JOIN

ILLEGAL CHILD TYPE Dieser Fehler tritt auf dem Parent Block eines Loches auf, wenn ein Block mit einem vom Parent nicht erlaubten Typ eingefügt wird. Dieser Fehler ist dafür verantwortlich, dass in Iteration 2 des Filters nicht erlaubte Blöcke teilweise nicht ausgeblendet wurden.

INVALID MIN OCCURENCES Dieser Fehler verhält sich ähnlich wie der *Missing Child Error*, nur dass dieser nicht auftritt, wenn einem Block ein weiterer fehlt um ausgewertet werden zu können, sondern, wenn die bestimmte Anzahl an Mindestblöcken, die eingesetzt werden müssen, nicht erreicht wird. Der Fehler tritt beispielsweise auf, wenn bei einem GROUP BY nichts angegeben wird, aber mindestens ein Wert erwartet wird, damit sortiert werden kann.

SUPERFLUOUS CHILD ERROR Dieser Fehler tritt auf, wenn ein unerwarteter Block AST eingefügt wird. So ein unerwarteter Block kann zum Beispiel eine Redundanz, wie ein doppelter

FROM Block sein. Dieser wäre dann ein Superfluous Child der Select Query. Dieser Fehler sollte im eingebauten Editor nicht auftreten können, da solche doppelten Blöcke erst gar nicht platziert werden dürfen. Das Umschalten in den Blockly Editor ermöglicht jedoch solche Blockkombinationen. So kann ein solcher Fehler ebenfalls im eingebauten Editor auftreten.

4.2 Filter Version 1: Direkter Aufruf aus dem HTML Template

Der Filter wurde entsprechend der in Kapitel 2 beschriebenen Funktionsweise umgesetzt. Die Filterfunktion `currentHoleMatchesBlock()` erstellt eine Liste der Fehler, die im potenziellen AST mit dem jeweiligen eingesetzten Block auftreten. Ist die Länge dieser Liste null, so gibt die Funktion `true` zurück und der entsprechende Block wird mit einem direkten Aufruf in `*ngIf` im HTML Template der *draggable-block-list.html* dann in der Seitenleiste angezeigt.

```
<ng-container *ngFor="let block of category.blocks">
    <div
        *ngIf="currentHoleMatchesBlock(block)"
        (mousedown)="startDrag($event, block)"
        class="block rounded"
    >
    <span>{{ block.displayName }}</span>
    </div>
</ng-container>
```

Listing 4.2: draggable-block-list.html Template mit direktem Aufruf der Filterfunktion

Es handelt sich hierbei um die einfachste Variante, die Blöcke in der Seitenleiste anzuzeigen. Ist das Ergebnis der Validierung für den Block ohne Fehler, so wird der Block angezeigt, Tritt ein Fehler bei der Validierung auf, dann wird der Block nicht angezeigt.

Hierbei ist zu beachten, dass aus der Liste der Fehler kategorisch “MISSING-CHILD” Fehler rausgefiltert werden, da diese lediglich besagen, dass an der Stelle im Baum ein Block fehlt. Das mag vielleicht wie ein wichtiger Fehler klingen, aber es fehlt beispielsweise auch ein Child, wenn eine Tabelle mit GROUP BY sortiert werden soll, aber noch kein Parameter zur Sortierung angegeben wurde. Dieses Fehlen allein bedeutet aber nicht, dass der GROUP BY Block generell nicht eingesetzt werden darf. Darüber hinaus existieren in der Seitenleiste diverse Blöcke, die im ursprünglichen Design unvollständig sind. Außerdem werden durch das Ignorieren aller “MISSING CHILD” Fehler auch alle anderen Löcher im Syntaxbaum ignoriert.

Diese erste Version des Filters funktioniert bereits in Specification Tests, hat aber Schwierigkeiten in der Praxis, da die Seite aufgrund von Cashing der Funktion im HTML und der

Funktionsweise des Validators tausende Aufrufe pro Sekunde gemacht werden und die Seite unter dieser Last nicht mehr vollständig Laden kann. Diese massiven Performanceprobleme sollen in Version 2 des Filters gelöst werden.

```
async currentHoleMatchesBlock(
 block: NodeDescription | FixedSidebarBlock
) {
    const validator = await this.validator$.pipe(first()).toPromise();
    const ast = this.peekSyntaxtree;
    const holeLocation = this._holeLocation.value;
    if (holeLocation === undefined) {
        return true;
    }
    const fillBlocks =
        block instanceof FixedSidebarBlock
        ? block.tailoredBlockDescription(ast)
        : [block];

    const possibleAst = ast.insertNode(holeLocation, fillBlocks[0]);
    const instertedNode = possibleAst.locate(holeLocation);

    const allErrors = validator.validateFromRoot(possibleAst);
    const errorList = validator
        .validateFromRoot(possibleAst)
        .getErrorsOn(instertedNode)
        .filter((error) => error.code != "MISSING_CHILD");

    return errorList.length == 0;
}
```

Listing 4.3: current-coderesource.service.ts in Filter Version 1

4.3 Filter Version 2: Wegfall des asynchronen Methodenaufrufs

Damit die Seite wieder geladen werden kann, wird eine neue Methode hinzugefügt, die sich um die Umwandlung des Validators in ein Promise kümmert und damit die Anzahl der Methodenaufrufe verringert. Das Ergebnis dieser neuen `currentHoleMatchesBlock()`Funktion wird dann in die umbenannte `currentHoleMatchesBlock2()` hineingegeben. Da-

durch läuft die vorherige Funktion nicht mehr asynchron und das caching findet nicht mehr im HTML, sondern in der Komponente statt.

```
async currentHoleMatchesBlock(
 block: NodeDescription | FixedSidebarBlock
) {
    const validator = await this.validator$.pipe(first()).toPromise();
    return this.currentHoleMatchesBlock2(block, validator);
  }
currentHoleMatchesBlock2(
 block: NodeDescription | FixedSidebarBlock,
 validator: Validator
) {
    //Validator wird hier geloescht, ansonsten bleibt Code wie zuvor
}
```

Listing 4.4: current-coderesource.service.ts in Filter Version 2 mit zusätzlicher Funktion

Für eine Verbesserung der Performance wird außerdem der direkte Aufruf der `currentHoleMatchesBlock()`Funktion durch die Verwendung einer neuen `isVisibleInSidebar()` Funktion in der *draggable-block-list.component.ts* ersetzt. In dieser neuen Funktion wird `currentHoleMatchesBlock2()`aufgerufen.

```
isVisibleInSidebar(block: NodeDescription | FixedSidebarBlock) {
    if (this.validator) {
      return this._currentCodeResource.currentHoleMatchesBlock2(
        block,
        this.validator
      );
    }
    return true;
}
```

Listing 4.5: isVisibleInSidebar() Funktion in draggable-blocklist-component.ts

```
<ng-container *ngFor="let block of category.blocks">
    <div
        *ngIf="isVisibleInSidebar(block)"
        (mousedown)="startDrag($event, block)"
        class="block rounded"
    >
        <span>{{ block.displayName }}</span>
    </div>
</ng-container>
```

Listing 4.6: Anpassung des HTML Template zur Benutzung von isVisibleInSidebar()

Diese Iteration brachte auch in der Praxis schon Ergebnisse, führte beim Testing aber zur Feststellung, dass trotzdem teilweise noch Blöcke angezeigt wurden, die eigentlich nicht erlaubt sein sollten. Des Weiteren war die Implementierung noch recht unperformant, da für jeden einzelnen Block der gesamte Syntaxbaum komplett mit allen enthaltenen Fehlern validiert werden musste. Das geschah nicht nur einmal, sondern jedes mal, wenn irgendeine Art von Event in der Anwendung ausgeführt wurde, auch zum Beispiel beim Bewegen der Maus.

4.4 Filter Version 3: Migration zur Nutzung von allowsChildType()

Wie bereits erwähnt können Fehler auch auf Parent Nodes auftreten. Deshalb wurde entschieden, in der weiteren Entwicklung des Filters einen anderen Ansatz zur Validierung der Blöcke zu nutzen, der bereits in ähnlicher Form in BlattWerkzeug implementiert ist. Die Einfärbung der Löcher und die Entscheidung, ob ein Block platziert werden darf oder nicht, erfolgt bereits über den Abgleich von erlaubten Typen von Children der Parent Nodes und dem Type des potenziellen Blöcken, die ein Loch füllen.

Um diese Logik auch für das Filtern der Seitenleiste nutzen zu können, werden in der `currentHoleMatchesBlock2()` Funktion Konstanten für den Eltern-Knotenpunkt des Lochs (`parentNode`) und den letzten Punkt des Orts, an dem der Block eingefügt werden soll (`dropLocation`) hinzugefügt. Um den Eltern-Knotenpunkt ermitteln zu können, muss zusätzlich aus der Position des Lochs die Postition des Parents des Lochs(`parentLocation`) bestimmt werden. Mit diesen neuen Werten wird im veränderten Validator der Typ des Eltern-Knotens bestimmt und daraufhin überprüft, ob der Typ des eingesetzten potenziellen Blocks aus der Seitenleiste in der Lücke erlaubt ist. Wenn die `parentNode` den Typ des als Kind eingesetzten Blocks erlaubt, wird dieser Block in der Seitenleiste angezeigt.

```
currentHoleMatchesBlock2(
    block: NodeDescription | FixedSidebarBlock,
    validator: Validator
  ): boolean {
    const ast = this.peekSyntaxtree;
    const holeLocation = structuredClone(this._holeLocation.value);

    if (holeLocation != undefined && holeLocation.length > 0) {
      const dropLocation = holeLocation[holeLocation.length - 1];
      const parentLocation =
        holeLocation.slice(0, holeLocation.length - 1);
      const parentNode = ast.locate(parentLocation);

      const fillBlocks =
        block instanceof FixedSidebarBlock
          ? block.tailoredBlockDescription(ast)
          : [block];

      const childBlockTypeName: QualifiedTypeName = {
        typeName: fillBlocks[0].name,
        languageName: fillBlocks[0].language,
      };
      return validator
        .getGrammarValidator(childBlockTypeName.languageName)
        .getType(parentNode)
        .allowsChildType(childBlockTypeName, dropLocation[0]);
    } else {
      return true;
    }
}
```

Listing 4.7: Filterfunktion mit Validator basierend auf ChildType

4.5 Filter Version 4: Umwandlung in Observable

Um die Performance des Filters weiter zu verbessern und die Stärken einer Umsetzung in Angular zu nutzen, wurden in der letzten Iteration des Filter Observables genutzt.

Die zuvor neu erstellten Konstanten wurden so aus der `currentHoleMatchesBlock2()` Funktion entfernt und, wie auch schon der Validator, in die Funktion hineingegeben. So müssen

keinerlei Werte mehr in der Filterfunktion ermittelt werden und es kommt zu einer sauberen Trennung von Funktionalitäten.

Die Erstellung von `currentHoleDropStep$` und `currentHoleLocationParent$` als Observables stellt außerdem sicher, dass `currentHoleMatchesBlock2()` nur ausgeführt wird, wenn es eine tatsächliche Änderung des ausgewählten Loches gibt. Das `currentHoleDropStep$` Observable ersetzt die `dropLocation`, das Observable `currentHoleLocationParent$` ersetzt die `parentNode` und die `parentLocation` wird mit der Verwendung der Observables nicht mehr benötigt.

Die Verbindung zur Seitenleiste wird geknüpft, indem auch in der *sidebar-blocks.ts*, die zuständig für die Blöcke in der Seitenleiste ist, ein Observable `isVisibleInSidebar$` erstellt wird. Dieses kombiniert mittels *RxJS* Operator `combineLatest()` die Datenströme der zuvor erstellten Observables und des Validators und führt mit Hilfe des `pipe()` Operators die `currentHoleMatchesBlock2()` Funktion aus.

```
public isVisibleInSidebar$: Observable<boolean> = combineLatest([
    this._currentCodeResource.validator$,
    this._currentCodeResource.currentHoleDropStep$,
    this._currentCodeResource.currentHoleLocationParent$,
  ]).pipe(
    map(([validator, currentHoleDropStep, currentHoleLocationParent])
    => { return this._currentCodeResource.currentHoleMatchesBlock2(
        this,
        validator,
        currentHoleDropStep,
        currentHoleLocationParent
      );
    })
);
```

Listing 4.8: Observable isVisibleInSidebar$

Das Observable wird dann auch im `*ngIf` der *draggable-block-list.html* eingesetzt und wird dort von der Angular Async Pipe entpackt und ausgewertet.

```
<ng-container *ngFor="let block of category.blocks">
    <div
        *ngIf="block.isVisibleInSidebar$ | async"
        (mousedown)="startDrag($event, block)"
        class="block rounded"
    >
    <span>{{ block.displayName }}</span>
```

```
8       </div>
9   </ng-container>
```

Listing 4.9: draggable-block-list.html Template mit Observable

Die Verbindung vom *current-coderesource.service.ts* zu *sidebar-blocks.ts*, den Blöcken in der Seitenleiste, wird hergestellt, indem der Service durch `SidebarDataService`, `FixedBlocksSidebar`, `FixedBlocksSidebarCategory`, `FixedSidebarBlock` gereicht wird.

5 Usability Evaluation

Unter *Evaluation* versteht sich die "systematische und möglichst objektive Bewertung eines geplanten, laufenden oder abgeschlossenen Projekts. Ziel ist es, spezifische Fragestellungen zu beantworten und/oder den Zielerreichungsgrad eines bestimmten Vorhabens zu erheben" [SB15, S. 24] Anhand von Abbildung 5.1 ist auf den ersten Blick eine Veränderung des Editors in BlattWerkzeug erkennbar.

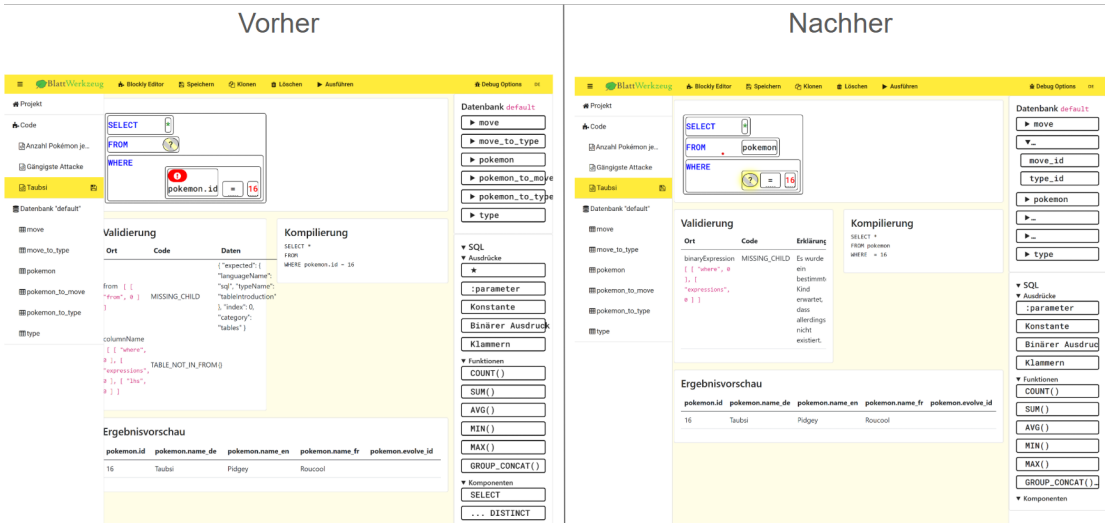

Abbildung 5.1: Bildschirmaufnamen vor und nach den Veränderungen dieser Arbeit

Bewertet man die Änderung an BlattWerkzeugs Usability anhand der Nielsen Heuristiken, so ist eine leichte Verbesserung in mehreren Bereichen zu erkennen:

1: Visibility of System Status Durch Hinzuügen eines Highlights um das ausgewählte Loch, wissen Nutzende nun immer, an welcher Stelle sie gerade den AST bearbeiten und für welche Stelle im AST die Seitenleiste gefiltert wird.

2: Match Between the System and the Real World Diese Heuristik bleibt unverändert.

3: User Control and Freedom Diese Heuristik bleibt unverändert.

4: Consistency and Standards Diese Heutistik hat eine leichte Verbesserung erfahren, da die Löcher, als Elemente, die bearbeitet werden können, nun auch angeklickt werden können.

5: Error Prevention Durch die Implementierung des Filters für potenzielle Blöcke werden mögliche Fehler durch das Einfügen falscher Elemente deutlich reduziert.

6: Recognition Rather than Recall Auch hier ist eine leichte Veränderung festzustellen, da Nutzende sich nicht mehr von alleine merken müssen, welche Blöcke erlaubt sind und welche Stelle im AST sie gerade bearbeiten.

7: Flexibility and Efficiency of Use Durch das Einschränken von Optionen zum Einsetzten von Blöcken konnte auch die Geschwindigkeit in der Syntaxbäume zusammengesetzt werden könnnen erhöht werden.

8: Aesthetic and Minimalist Design Diese Heuristik konnte durch die Erfüllung der nichtfunktionalen Anforderungen etwas besser erfüllt werden, da der Editor nun aesthetisch ansprechender wirkt.

9: Help Users Recognize, Diagnose, and Recover from Errors Auch hier konnte die Usability etwas verbessert werden, da die Validierungsausgabe nun über eine menschenlesbare Erklärung der Error Codes verfügt.

10: Help and Documentation Diese Heuristik bleibt unverändert.

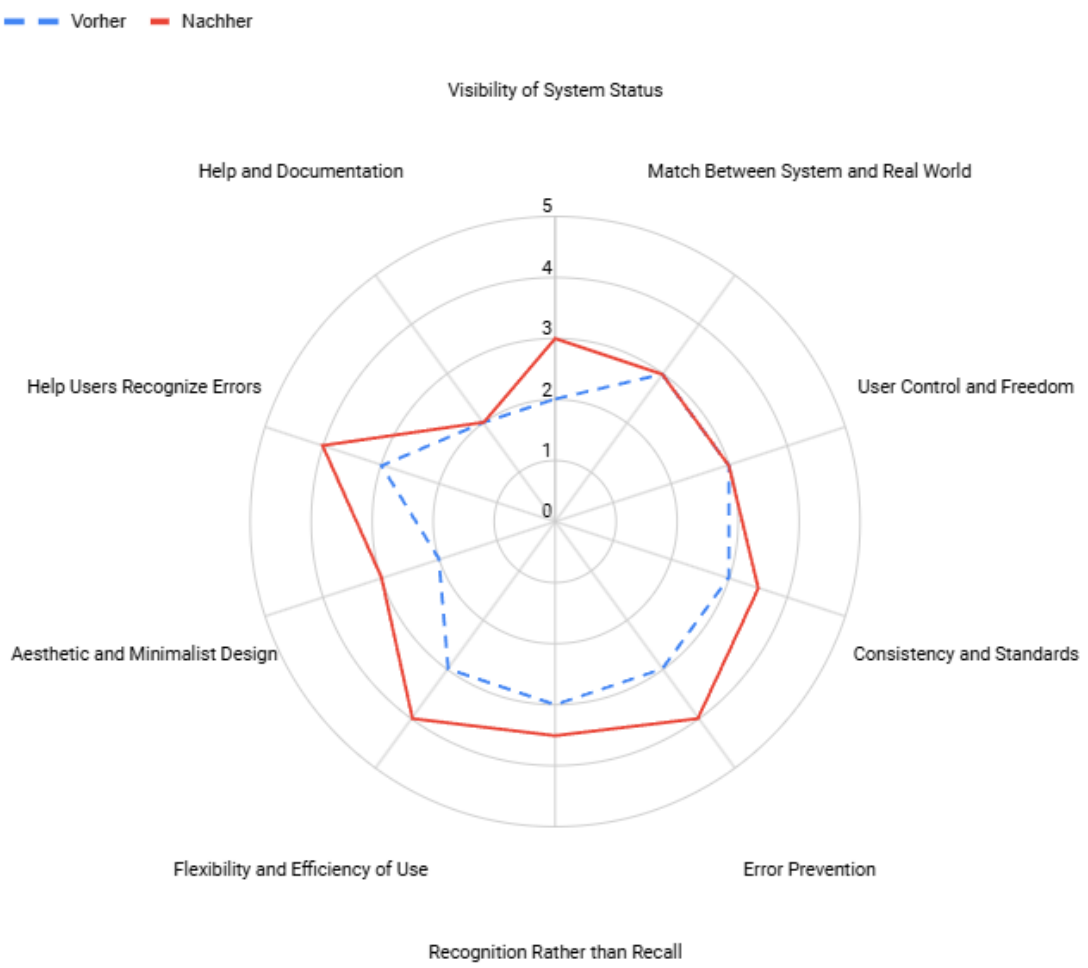

Abbildung 5.2: Die Verbesserung der Usability laut Nielsen Heuristiken

Trotz der Erfolgreichen Entwicklung des Filters sind während der Analyse von BlattWerkzeugs Usability folgende weitere Schwachstellen aufgefallen:

Ohne ausgiebige Nutzertests ist es sehr schwierig einzuschätzen wie stark Nutzende Schüler*innen tatsächlich von der Software selbst geleitet werden müssen, da der gewünschte Anwendungsfall immer im Zusammenhang mit schulischem Unterricht stattfinden soll. Wie genau dieses Zusammenspiel aussieht ist nicht abzusehen, da sich dazu bisher nur theoretische Überlegungen gemacht wurden. So stellen sich für BlattWerkzeug entwickelnden Fragen wie:

Werden im Unterricht Programmierprinzipen im Vorfeld erklärt? Wissen Nutzende, wie bestimmte Funktionen in SQL arbeiten? Wird Nutzenden erklärt was ein Parameter ist in SQL? Wird Nutzenden der Unterschied zwischen Datenbankblöcken und Sprach Blöcken erklärt?

Eine weitere Herausforderung, die tatsächlich den Umgang mit dem entwickelten Filter betrifft, ist, dass in BlattWerkzeug jederzeit während der Nutzung in den Blockly Editor umgeschaltet werden kann. Auf diesen hat der implementierte Filter keinen Einfluss, was bedeutet, dass es weiterhin möglich ist beliebige Blöcke an beliebigen Stellen einzusetzen und somit Fehler zu provozieren.

Außerdem musste bei der Entwicklung festgestellt werden, dass sich das Filtern der Datenbank Blöcke und damit die Vermeidung von semantischen Datenbankspezifischen Fehlern wie TABLE NOT IN FROM nicht mit dem entwickelten Filter abgeden lassen.

6 Fazit

Die vorliegende Arbeit hatte zum Ziel die User Experience bei der Nutzung der Lernumgebung BlattWerkzeug zu verbessern. Dazu sollte ein Filter entwickelt werden, der Erweiterungen von Syntaxbäumen auf Zulässigkeit prüft. Um zu verstehen, wie genau die User Experience verbessert werden kann musste in Kapitel 2 zunächst erarbeitet werden, was unter Usability verstanden wird, welche Regeln für gutes UX Design gelten und wie UX gemessen werden kann. Außerdem war es erforderlich sich mit BlattWerkzeug und den Ansätzen der Lernumgebung auseinanderzusetzen, damit sich der zu entwickelnde Filter angemessen in das bereits vorhanden System einfügt. In Kapitel 3 wurde zunächst BlattWerkzeug genauer betrachtet und festgestellt inwiefern die erarbeiteten UX Regeln auf das System anwendbar sind. Des weiteren wurden anhand der Zielgruppenanalyse Personas von Nutzenden erstellt um deren Bedürfnisse und Anforderungen genauer definieren zu können. Die anschließende Anforderungsanalyse stellte die funktionalen Anforderungen *FA-1 Blockfilter*, *FA-2 Lückenauswahl* und nicht-funktionalen Anforderungen *NA-1 Lesbarkeit*, *NA-Übersicht. NA-3 Verständlichkeit* auf. Kapitel 4 ging auf die iterative Implementierung des Filters zur Erfüllung der funktionalen Anforderungen ein. Nach anfänglichen Specification-Tests zur genaueren Eingrenzung der Funktionsweise des Filters, wurde dieser, wie geplant mit einer Kombination aus Validator, Hole Location, AST und dem einzusetzendem Block umgesetzt. In der ersten Iteration noch recht primitiv und kaum lauffähig, wurde die Performance stetig verbessert. Als diese Art der Implementierung an ihre Grenze stieß, wurde für die letzte Iteration des Filters eine andere, bereits in BlattWerkzeug implementierte Methode der Validierung genutzt. Diese Variante machte sich die Stärken der gewählten Frontendtechnologie Angular zu nutzen indem sie Observables mit in die Nutzung einbezog. Zusätzlich wurden die technisch weniger komplexen Anforderungen FA-1 und FA-3 umgesetzt, in der Implementierung aber nicht genauer behandelt werden. Die Evaluation der Usability in Kapitel 5 zeigt eine Verbesserung der User Experience laut Nielsen Heuristiken, auch wenn nicht alle in Kapitel 3 gesetzten Anforderungen erfüllt wurden.

6.0.1 Ausblick

Als ersten Schritt ist es sinnvoll den im Zuge dieser Arbeit entwickelten Filter weiterzuentwickeln. Das meint sowohl einen Feinschliff der vorhandenen Umsetzung, als auch die Erweiterung des Umfangs, damit auch semantische Fehler verhindert werden können. Dazu muss der Filter auf die hinterlegten Datenbanken zugreifen können und deren Daten verarbeiten können, sodass beispielsweise erkannt werden kann, ob ein bestimmter Wert überhaupt in einer abgefragten Tabelle enthalten sein kann.

Des weiteren sollte darüber nachgedacht werden, inwiefern Blockly noch eine sinnvolle Ergänzung in BlattWerkzeug darstellt. Wenn der zusätzliche Editor nicht entfernt wird, dann sollten Nutzende zumindest direkter darauf hingewiesen werden, dass die Beste Nutzungserfahrung mit dem anwendungseigenen Editor gemacht werden kann.

Schlussendlich ist es unerlässlich mit dem aktuellen Prototypen von BlattWerkzeug Nutzungstests in Form von Fokusgruppen und Befragungen unter Hinblick auf Methoden aus DIN EN ISO 9241 durchzuführen. So können praktische Eindrücke der Nutzung gewonnen werden und Anforderungen für gezielte Verbesserungen aufgestellt werden.

Literatur

[Di +22] Davide Di Ruscio u. a. "Low-code development and model-driven engineering: Two sides of the same coin?" en. In: *Software and Systems Modeling* 21.2 (Apr. 2022), S. 437–446. ISSN: 1619-1374. DOI: 10.1007/s10270-021-00970-2. URL: https://doi.org/10.1007/s10270-021-00970-2 (besucht am 25. 09. 2024).

[DIN18] DIN. *DIN EN ISO 9241-11:2018-11, Ergonomie der Mensch-System-Interaktion_- Teil_11: Gebrauchstauglichkeit: Begriffe und Konzepte (ISO_9241-11:2018); Deutsche Fassung EN_ISO_9241-11:2018*. de. 2018. DOI: 10.31030/2757945. URL: https://www.dinmedia.de/de/-/-/279590417 (besucht am 26. 09. 2024).

[DIN19] DIN. *DIN EN ISO 9241-220:2020-07, Ergonomie der Mensch-System-Interaktion_- Teil_220: Prozesse zur Ermöglichung, Durchführung und Bewertung menschzentrierter Gestaltung für interaktive Systeme in Hersteller- und Betreiberorganisationen (ISO_9241-220:2019); Deutsche Fassung EN_ISO_9241-220:2019*. de. 2019. DOI: 10.31030/2851768. URL: https://www.dinmedia.de/de/-/-/289443385 (besucht am 26. 09. 2024).

[DIN20] DIN. *DIN EN ISO 9241-210:2020-03, Ergonomie der Mensch-System-Interaktion_- Teil_210: Menschzentrierte Gestaltung interaktiver Systeme (ISO_9241-210:2019); Deutsche Fassung EN_ISO_9241-210:2019*. 2020. DOI: 10.31030/3104744. URL: https://www.dinmedia.de/de/-/-/313017070 (besucht am 28. 09. 2024).

[Gooa] Google. *FAQs | Google AppSheet*. en. URL: https://about.appsheet.com/faq/ (besucht am 25. 09. 2024).

[Goob] Google. *What is Blockly*. en. URL: https://developers.google.com/blockly/guides/get-started/what-is-blockly (besucht am 25. 09. 2024).

[Mos12] Christian Moser. *User Experience Design: Mit erlebniszentrierter Softwareentwicklung zu Produkten, die begeistern*. de. X.media.press. Berlin, Heidelberg: Springer, 2012. ISBN: 978-3-642-13362-6 978-3-642-13363-3. DOI: 10.1007/978-3-642-13363-3. URL: https://link.springer.com/10.1007/978-3-642-13363-3 (besucht am 21. 01. 2025).

[Nie94] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. en. Apr. 1994. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (besucht am 29. 01. 2025).

[Res+09] Mitchel Resnick u. a. "Scratch: programming for all". en. In: *Communications of the ACM* 52.11 (Nov. 2009), S. 60–67. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1592761.1592779. URL: https://dl.acm.org/doi/10.1145/1592761.1592779 (besucht am 25. 09. 2024).

[Rie] Markus Riemer. *Core Concepts — BlattWerkzeug 0.1 documentation*. Manual. URL: https://manual.blattwerkzeug.de/en/latest/concepts.html (besucht am 27. 01. 2025).

[Rie16] Markus Riemer. "BlattWerkzeug - Eine datenzentrierte Entwicklungsumgebung für den Schulunterricht". Deutsch. Master-Thesis. FH Wedel, 2016.

[RxM] RxMarbles. *RxMarbles: Interactive diagrams of Rx Observables*. URL: https://rxmarbles.com/ (besucht am 01. 10. 2024).

[SB15] Florian Sarodnick und Henning Brau. *Methoden der Usability Evalution : Wissenschaftliche Grundlagen und praktische Anwendung*. Hogrefe Verlag GmbH & Co. KG, Dez. 2015. ISBN: ISBN 9783456855974. DOI: 10.1024/85597-000. URL: https://elibrary.hogrefe.com/book/10.1024/85597-000.

[Scr] Scratch. *Scratch - About*. de-DE. URL: https://scratch.mit.edu/ (besucht am 25. 09. 2024).

[Swe05] John Sweller. "Implications of Cognitive Load Theory for Multimedia Learning". In: *The Cambridge Handbook of Multimedia Learning*. Hrsg. von Richard Mayer. Cambridge Handbooks in Psychology. Cambridge: Cambridge University Press, 2005, S. 19–30. ISBN: 978-0-511-81681-9. DOI: 10.1017/CBO9780511816819.003. URL: https://www.cambridge.org/core/books/cambridge-handbook-of-multimedia-learning/implications-of-cognitive-load-theory-for-multimedia-learning/F5F9582CB12C6781FA9C61F6B45 (besucht am 01. 10. 2024).

[tap24] tapanm-MSFT. *Was ist Power Apps? - Power Apps*. de-de. Aug. 2024. URL: https://learn.microsoft.com/de-de/power-apps/powerapps-overview (besucht am 25. 09. 2024).

[Yab20] Jon Yablonski. *Laws of UX: using psychology to design better products & services*. eng. First edition. Beijing Boston Farnham Sebastopol Tokyo: OReilly, 2020. ISBN: 978-1-4920-5531-0.

Abbildungsverzeichnis

Listings

Glossar

CTL Cognitive Load Theory. 1

LCDP Cognitive Load Theory. 11

RxJS Reactive Extensions Library for JavaScript. 18

UI User Interface. 9

UX User Experience. 2

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

Hamburg, 29. Januar 2025 Julia Schmaglowski