Ein Bild, das Text, Schrift, Logo, Grafiken enthält.

KI-generierte Inhalte können fehlerhaft sein.

Bachelorarbeit

Internationale Hochschule Fernstudium

Studiengang: B. Sc. Wirtschaftsinformatik

Die Vorteile der High-Code Entwicklung im Vergleich zur Low-Code Entwicklung

Evsin Rahmiev Matrikelnummer: 32105477

Akademiestr. 6

68159 Mannheim

Betreuungsperson: Prof. Dr. Thorsten Fröhlich 02.05.2025

Abstract

Im Bereich der Softwareentwicklung bietet sich neben der Möglichkeit, eine Software konventionell über High-Code Entwicklung mittlerweile auch die Option, über Low-Code zu entwickeln. Beide Ansätze haben unterschiedliche Vor- und Nachteile, je nachdem, was für Ziele hinter der Entwicklung stehen. Die vorliegende Arbeit hat die Aufgabe, die Frage zu beantworten, welche Vorteile die High-Code Entwicklung im Vergleich zu Low-Code anbietet. Zum Zweck der Arbeit und zur Beantwortung der Forschungsfrage wurde ein Case Study durchgeführt, bei der eine einfache Anwendung zum Deutschlernen einerseits über die etablierten Webentwicklungstechnologien HTML, CSS und JavaScript, also High-Code Werkzeuge, und andererseits über die Low-Code Entwicklungsanwendung *Zoho Creator* implementiert wurde. Es wurde untersucht, welche der beiden Entwicklungsansätze besser geeignet ist, um die konzipierte Lernanwendung mit ihren geplanten Funktionalitäten wie z. B. Verlinkung zwischen den Seiten der Anwendung, Manipulation der UI-Elemente, die grafische und funktionelle Anpassung von Formularen umzusetzen. Die Ergebnisse zeigen, dass sich die geplanten Anwendungskomponenten über High-Code Entwicklung näher an die Anforderungen implementieren lassen, weil diese Standardwebentwicklung mehr Anpassungsmöglichkeiten bietet. Über Low-Code hat man eher den Vorteil, dass man mit weniger Programmierkenntnissen eine Anwendung bauen kann, die dafür aber technisch weniger vielfältig ist.

**Schlüsselwörter**: High-Code, Low-Code, Softwareentwicklung, Anwendung

Abstract

Apart from programming an application from scratch suing conventional high-code development, the field of software development also offers the opportunity of developing an application using low-code development. Both approaches have different advantages and disadvantages depending on the goals oft he development. The present project aims to answer the question of which advantages high-code development offers compared to low-code development. Fort he purpose oft he project and in ordert to answer the research question, a case study was conducted where a simple application for learning German was developed by using established web development technologies like HTML, CSS and JavaScript on one hand and the low-code development platfrom Zoho Creator on the other. The experiment aimed to discover which one oft he two development methods is more suitable for programming the conceived learning application and its planned features such as links between the application’s pages, manipulation of ist ui-elements and the graphical i.e. visual and functional customization of forms. The results show that development through high-code allows for the application’s requirements tob e met to a higher level because standard development offers more customization opportunities. Low-code development offers more of an advantage in terms of coding an application with less programming skills, which in terms makes the application less technically diverse.

**Key words**: high-code, low-code, software development, application

Inhaltsverzeichnis

AbstractI

I. AbkürzungsverzeichnisII

II. AbbildungsverzeichnisIII

III. TabellenverzeichnisIV

1. Einleitung1

1.1 Forschungsziel4

1.2 Forschungsfragen4

1.3 These4

1.4 Wert und Zielpublikum4

1.5 Anwendungsbereich und Einschränkungen4

1.6 Struktur des Dokuments4

2. Theoretische Fundierung1

2.1 Terminologie und Definitionen4

2.2 Relevante Theorien und Konzepte4

3. Forschungsdesign2

3.1 Anforderungsanalyse4

3.2 Spezifikation der Anforderungen4

3.3 Technologie4

3.4 Analyse4

3.5 Fehlerbewertung4

4. Forschungsergebnisse und DIskussion3

5. Fazit15

IV. Literaturverzeichnis16

I. Abkürzungsverzeichnis

1. HC – High-Code1

3. LC/ NC – Low-Code / No-Code3

4. LCAP – Low-Code-Applikationsplattform2

4. lDE – Integrated Development Environment2

7. DOM – Document Object Model1

8. OOP – Object Oriented Programming1

9. ROI – Return on Investment1

9. VSC – Visual Studio Code1

II. Abbildungsverzeichnis

Abb. 1: Iron Triangle2

Abb. 2: Magisches Viereck2

Abb. 3: Tagesaufwand für die Fehlerbehebung2

III. Tabellenverzeichnis

Tab. 1: Funktionale Anforderungen13

Tab: 2: Nicht-funktionale Anforderungen13

Tab: 3: Technische Anforderungen13

1. Einleitung

Die Technologie der Low-Code/No-Code Entwicklung bietet einen alternativen Ansatz zur Softwareentwicklung, mithilfe derer die Entwicklungszeit für eine ausgereifte Softwarelösung drastisch reduziert werden kann. Dies ist möglich dank der Art und Weise der LC/NC-Entwicklung, bei der Systemkomponenten und Module zum Teil durch das einfache Auswählen bzw. Drag-and-Drop von vorprogrammierten UI-Elementen wie z. B. Auswahlfelder, Tabellen oder Grafiken implementiert werden können. Dies stellt eine Erleichterung im Vergleich zu der bereits etablierten und mehr bekannten High-Code Entwicklung, bei der die Umsetzung ähnlicher Systemkomponenten in jedem Fall das Schreiben von Programmcode erfordert. Diese Unterschiede der beiden Programmieransätze lassen sich bei der Entwicklung von LexiCode, eine Anwendung zum Deutschlernen, auch bemerkbar. Die Anwendung, die grob beschrieben eine reine Frontend Anwendung ist und unter anderem Theorie zu den grammatikalischen Grundlagen der deutschen Sprachen übermitteln und diese durch interaktive Übungsaufgaben überprüfen soll. LexiCode kann in der Theorie sowohl über HC- als auch LC/NC-Entwicklung umgesetzt werden. Die Fragen, die in der Praxis allerdings beantwortet werden müssen, sind, welcher der beiden Ansätze ermöglicht, dass die konzipierten Systemmerkmale näher an die Anforderungen programmiert werden können und welcher der Ansätze bietet breiter Anpassungsmöglichkeiten für LexiCode an.

1.1 Forschungsziel

Die vorliegende Arbeit hat die Aufgabe, nachzuweisen, dass die klassische HC-Entwicklung besser geeignet ist, um die Anforderungen an LexiCode technisch umsetzen zu können, also die geplanten Systemkomponenten im Sinne von UI-Elementen und Funktionalitäten möglichst nah an die geplanten Komponenten zu halten. Es wird erforscht, welcher der beiden Entwicklungsansätze eine Umsetzung ermöglicht, die die Anwendung LexiCode möglichst interaktiv, also für Interaktionen zwischen Nutzern und der Anwendung sorgt, visuell und funktionell anpassungsfähig macht, eine dynamische Navigation zwischen den Seiten der Anwendung ermöglicht, eine Manipulation der UI-Elemente anhand der Nutzeraktivitäten bereitstellt, den Einbau von inzeiligen Eingabefeldern erlaubt und sitzungsunabhängig Nutzerdaten im Browser speichern kann.. Darüber hinaus wird geprüft, welche der beiden Ansätze bessere Anpassungsmöglichkeiten bietet, wenn die Anwendung für mobile Endgeräte erweitert wird.

1.2 Forschungsfragen

Um dieses Ziel zu erreichen, werden die folgenden Forschungsfragen beantwortet:

**1. Können über HC- oder LC/NC-Entwicklung Hyperlinks angelegt werden, die die User auch vom Inhalt einer Seite, also nicht nur von der globalen Navigation heraus, auf einer anderen Seite innerhalb der Anwendung umleiten können?**

Anwendungen verfügen normalerweise über eine globale Navigation, die die Benutzer auf die Hauptseiten der Anwendung verleiten können. Oftmals werden Links zu anderen Seiten auch im Inhalt einer Seite eingebaut, damit nicht alle verwandten Inhalte auf der gleichen Seite stehen und somit eine Übersichtlichkeit erreicht wird.

**2. Über welchen der Entwicklungsansätze können die Seitenelemente effektiver oder überhaupt manipuliert werden, um mehr Nutzungseffekte zu erreichen?**

Bei der HC-Entwicklung ist in der Programmiersprache JavaScript das Konzept bzw. Werkzeug des DOMs bekannt, welches die Möglichkeit bietet, dass an bestehende Elemente der der Anwendung neue Elemente wie Überschriften, Zeilen oder Bilder angeknüpft oder bestehende Elemente gelöscht werden (Flanagan, 2020, S. 416). Eine Unterfrage zu dieser Forschungsfrage ist, ob die gleiche Manipulation bei LC/NC vorhanden ist.

**3. Ist die Auswahl an Anpassungsmöglichkeiten für Webformulare breiter bei HC- oder LC/NC-Anwendungen?**

Auf Webanwendungen sind häufig Formulare zu finden. Diese erfüllen die gleiche Aufgabe wie Formulare in Papierform, mit dem Unterschied, dass diese digitalen Formulare u.a. die abgefragten Daten schneller und direkt an die abfragende Instanz schicken können. Die Anpassungsmöglichkeiten sollen die optische Gestaltung der Formulare und ihre Funktionalität umfangen.

**4. Lassen sich sowohl über HC als auch über LC/NC inzeiligen Eingabefelder einbauen und wie gut lassen sich diese anpassen?**

Beim Sprachenlernen sind Übungsaufgaben üblich, bei denen ein Wort, mehrere Worte oder komplette Satzteile ausgelassen werden und stattdessen sich Eingabefelder auf deren Plätzen befinden. Solche Felder sind eine Anforderung an LexiCode und lassen die Frage stellen, ob sich diese über HC oder LC/NC einbauen lassen und wenn ja, ob sie angpeasst werden können wie z. B. in der Breite, in der Höhe, in der Randdichte oder die Hintergrundfarbe.

**5. Können Funktionen umgesetzt werden, die benutzerdefinierte Inhalte lokal und sessionunabhängig speichern können?**

Bei der Arbeit mit einer Anwendung werden oft Daten in der Anwendung angegeben, die nach einer kurzen oder sogar längeren Zeit wieder abrufbar sein sollten. Obwohl eine Datenbankanbindung eine gängige Lösung für Datenspeicherung ist, ist für die Speicherung von Daten aller Arten nicht zwingend erforderlich. Antworten zu Übungsaufgaben oder Notizen können z. B. auf dem Browser gespeichert werden und sind trotzdem wieder in der Anwendung zu finden.

1.3 These

Die gestellten Forschungsfragen setzen Erwartungen an die Forschung sowie eine klare Vorstellung für die Endergebnisse. Unabhängig von diesen Erwartungen kann die Aussage formuliert werden, dass HC-Entwicklung aus der Sicht der breiten Verfügbarkeit von Werkzeugen zur Umsetzung der jeweiligen Anforderungen einen einfacheren Entwicklungsprozess zulässt.

1.4 Wert und Zielpublikum

Die Ergebnisse dieser Forschung leisten einen Beitrag zur Debatte über den effektiveren Entwicklungsansatz. Die verteidigte Ansicht, dass die LC/NC-Entwicklung die Zukunft der Softwareentwicklung wäre, beschränkt sich nur auf den Aspekt der Benutzerfreundlichkeit von LC/NC-Entwicklungsplattformen bzw. die Einfachheit, über solche Plattformen eine Software ins Leben zu rufen. Zwei Aspekte, die allerdings bei dieser Ansicht oft ignoriert werden, sind die Kompatibilität von LC/NC-Anwendungen mit anderen Systemen und die Verfügbarkeit von Werkzeugen, die weitere Konfigurations- und Anpassungsmöglichkeiten bieten oder auch für Wiederverwendbarkeit des entwickelten Programmcodes, Skalierbarkeit und Wartung ermöglichen. Dafür werden Bibliotheken, Frameworks und IDEs eingesetzt. Die Ergebnisse der Forschung sind für Softwareentwickler interessant und zwar unabhängig von der Erfahrung, Ausbildung, Branche oder Fachgebiet.

1.5 Anwendungsbereich und Einschränkungen

Die Relevanz der Forschungsergebnisse schränkt sich auf kleinere bis mittelgroßen Anwendungen ein, weil dies, zumindest im Rahmen dieser Forschung, die vorgesehene Größe der Anwendung LexiCode ist. Ferner sind die Ergebnisse für Personen interessant, die über wenig Erfahrung als Softwareentwickler verfügen, in eher kleinere Teams oder sogar allein die Entwicklungsarbeiten durchführen werden, mit weniger hardware- und softwaretechnischen Ressourcen arbeiten werden, eine überschaubare Anwendung entwickeln möchten und sich überlegen, ob sie sich für HC oder LC/NC entscheiden sollen. Die Ergebnisse der Forschung sind auch für Lernanwendungen relevant, die andere Inhalte als eine Fremdsprache übermitteln wollen.

Die Forschung befasst sich nicht mit Prozessen und Diensten, die die Speicherung oder das Abrufen von Anwendungs- oder Nutzerdaten von einem System oder Schnittstelle, die im Hintergrund der Anwendung laufen, wie z. B. eine Datenbank, ein Server oder eine Schnittstelle, weil die Umsetzung solcher Lösungen zusammen mit den in den vorherigen Kapiteln beschriebenen Funktionalitäten den vorgesehenen Rahmen dieser Forschung übersprengen würde. Insofern sind die Ergebnisse der Forschung eher für Frontend-Lösungen relevant.

1.6 Struktur des Dokuments

In dem darauffolgenden Teil des Dokuments werden die konkreten Schritte abgewickelt, die zum Forschungsergebnis führen. Im zweiten Kapitel wird der aktuelle Stand der Literatur zum Forschungsthema analysiert, um herauszufinden, wie LC/NC-Entwicklung momentan wahrgenommen wird, inwiefern der Ansatz im Vergleich zu HC-Entwicklung bevorzugt wird, für was für einen Typ Anwendungen er angewendet wird und von was für Entwickler. Darüber hinaus werden auch wichtige Begriffe im Bereich der Softwareentwicklungen genannt und erklärt. Im dritten Kapitel wird die Methodik bzw. das Forschungsdesign erläutert und es wird näher darauf eingegangen, wie die Idee von LexiCode einerseits mit HC und andererseits mit LC/NC umgesetzt wird und mit welchen konkreten Technologien bei beiden Ansätzen gearbeitet wird. Im darauffolgenden Kapitel, im 4. Kapitel, werden die Forschungsergebnisse vorgestellt und genau erläutert, welche der angeforderten Komponenten von LexiCode mit welchem der beiden Entwicklungsansätzen inwiefern oder überhaupt umgesetzt werden konnten. Im nächsten Abschnitt, der Diskussionsabschnitt, werden die Ergebnisse der Forschung kurz zusammengefasst, interpretiert und mit den Forschungserwartungen verglichen. Zusätzlich werden die Beschränkungen der Forschung besprochen. Im letzten Kapitel und abschließenden Kapitel wird auf die Forschung kritisch reflektiert und es werden Ideen für weiterführende Forschungen empfohlen.

2. Theoretische Fundierung

Im Folgenden werden relevante Beiträge zum Diskurs über LC/NC-Entwicklung vorgestellt. Die Beiträge unterscheiden sich einerseits nach der Umsetzung, weil manche von denen konkrete Fallstudien präsentieren, während andere rein theoretisch sind und nur die Literatur von anderen Autoren bzw. Forschern, die sich mit dem Thema befasst haben, vergleichen. Die berücksichtigte Literatur stammt aus den vergangenen drei Jahren, da die LC/NC-Entwicklung erst in dieser Zeitspanne etwas mehr an Bedeutung zugewonnen hat.

Die Auswahl der Beiträge erfolgte nach einer einfachen Suche nach dem Begriff *Low-Code Entwicklung* und ergab auch Quellen, die sich zwar auf No-Code Entwicklung konzentrieren, allerdings über KI-Chatbots und andere ähnliche Technologien. Da die Entwicklung mittels solcher Technologien keinen Einsatz in dieser Forschung finden, wurden solche Quellen auch nicht berücksichtigt.

2.1 Terminologie und Definitionen

Nachfolgend werden einige wichtige Begriffe definiert, die im Laufe der Forschung auftreten werden.

**Citizen Developer**: Anwender, die über formale Programmierkenntnisse nicht verfügen (Di Ruscio et al., 2022, S. 437).

**LCAP**: Anwendungen, die durch visuelles Zusammenfügen vorgefertigter Software-Bausteine erstellt und individuell angepasst werden können (Baumgarten et al., 2014, S. 1214).

**Time to Market**: die Zeit, die vergeht, bis die Entwicklung einer Produktidee oder ein Serviceangebot reif genug ist, damit eine Platzierung des Produktes bzw. der Dientleistung am Markt erfolgen kann (Gabler Wirtschaftslexikon, 2025)

**Minimum Viable Product (MVP)**: der minimalst mögliche Funktionsumfang eines Produkts oder einer Dienstleistung. Das Produkt / die Dienstleistung werden erst dann veröffentlicht, wenn diese Entwicklungsstufe erreicht wird, was an dem Treffen zwischen minimalem Aufwand und qualitativem Feedback zu erkennen ist (Gabler Wirtschaftslexikon, 2025).

**Earned-value-Ansatz**: Softwareentwicklungsansatz, bei dem die einzelnen Anforderungen mit *Value-Points* gewichtet werden, anstatt gleich behandelt zu werden. Die Anforderungen mit den meisten Wertpunkten werden vorgezogen, während die mit den wenigsten Punkten zurückgestellt werden. (ähnlich zu der agilen Softwareentwicklung, bei der Anforderungen je nach ihrem Wert für den Kunden abgearbeitet werden) (Sneed & Jungmayr, 2011, S. 192).

**Value-Driven Software-Engineering (wertgetriebenes Software-Engineering)**: Ein Softwareentwicklungsprojekt soll wirtschaftlich gerechtfertigt werden, indem die Kosten mit dem Nutzen eines Produkts abgeglichen werden (Sneed & Jungmayr, 2011, S. 192).

2.2 Relevante Theorien und Konzepte

Ein Konzept, welches für die vorliegende Forschung Relevanz hat, ist das magische Dreieck des Projektmanagements bzw. der sog. *Iron Triangle* (Atkinson, 1999, S. 338). Dieses Dreieck umfasst die Dimensionen Kosten, Zeit und Qualität und hat eine Bedeutung für die vorliegende Forschung, da der Vergleich der beiden Entwicklungsansätze auch ein Projekt darstellt, für welches Ressourcen wie Zeit eingesetzt werden und eine gewisse Qualität des Endergebnisses erwartet wird:

Abb. 1: Iron Triangle.

Ein Bild, das Reihe, Screenshot, Dreieck, Astronomie enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung auf der Basis von Atkinson, 1999.

Baumgarten et al. (2024) haben für ihre Forschung ein Modell eingesetzt, welches auf dem *Iron Triangle* von Atkinson basiert und dieses sogar erweitert. Sie bilden in ihrer Forschung ein sog. Magisches Viereck ab, welches neben den drei Dimensionen des Dreiecks noch die vierte Dimension der Flexibilität einschließt und somit ein sog. *magisches Viereck* darstellt (Baumgarten et al., 2024, S. 1213; S. 1217):

Abb. 2: Magisches Viereck



Quelle: Eigene Darstellung auf der Basis von Baumgarten et al., 2024.

Die von Baumgarten et al. (2024) vierte Dimension der Flexibilität ist auch bei der Entwicklung von LexiCode signifikant, weil darunter Aspekte der Anwendung verstanden werden, die für den technischen Ausbau und die Erweiterung der Anwendung sorgen durch z. B. Integration mit Systemen von Drittanbietern, Skalierbarkeit oder mehrere verschiedene Wege, eine Funktionalität auf dem Frontend umzusetzen. Systemübergreifende Integration und Skalierbarkeit sind Aspekte, die im Rahmen dieser Forschung nicht für die Anwendung LexiCode angestrebt wurden, aber nur zum Zweck der Erläuterung der Dimension der Flexibilität erwähnt wurden. Die von Baumgarten et al. eingesetzte Abbildung des magischen Vierecks veranschaulicht die Ergebnisse der literarischen Forschung, die sie durchgeführt haben und bei der sie unterschiedliche Beiträge zum Thema LC/NC-Entwicklung berücksichtigt haben, wie z. B. Foren, in denen Low-Code bezogene Diskussionen stattfinden, eine Forschung, die nach dem Grund des Einsatzes von Low-Code fragt, eine Forschung, die die Treiber und Bremsfaktoren einer LCAP-Einführung untersucht und eine Forschung, die sich mit den technischen Herausforderungen einer LCAP-basierten Applikationsentwicklung beschäftigt (S. 1217). Je nachdem, wie die Auswirkungen der LC/NC-Entwicklung von den Autoren dieser Forschungen wahrgenommen worden sind, haben Baumgarten et al. die identifizierten Praxiserfahrungen der entsprechenden Dimension auf dem Viereck zugeordnet und mit einem Faktor von „+1“ auf die Dimension abgebildet, wenn die Praxiserfahrung als Verbesserung zur HC-Entwicklung wahrgenommen wurde. Mit „-1“ wurde eine entgegengesetzte Praxiserfahrung abgebildet und mit „0“ eine neutrale (Baumgarten et al., 2014. S. 1217). Auf Abbildung 2. sind die Ergebnisse der von Baumgarten et al. referenzierten Forschungen geschildert, die auf Zugewinne in der Dimension der Zeit hindeuten. Dabei haben sich Baumgarten et al. darauf konzentriert, wie lang die Iterationszyklen bei der Entwicklung einer LCAP sind, wie schnell das MVP und wie schnell das Time To Market erreicht werden kann (S. 1218). Die Ergebnisse der vorliegenden Forschung werden im späteren Kapitel über die Ergebnisse ebenfalls visuell auf dem magischen Viereck dargestellt, damit veranschaulicht werden kann, in welchen Dimensionen die LC/NC- und in welchen Dimensionen HC-Version von LexiCode zu- oder abgewonnen hat.

Ein weiterer theoretischer Hintergrund, der die vorliegende Forschung unterstützt, äußert sich in der Betrachtung der Messung eines Softwareentwicklungsprojektes, um die (weitere) Durchführung des Projektes überhaupt zu rechtfertigen. Sneed und Jungmayr (2011, zitiert nach Boehm, 2000) heben die Vorteile des Earned-value-Ansatzes hervor, bemängeln jedoch, dass bei diesem Ansatz die Entscheidung nicht unterstützt wird, ob ein (Teil)Projekt überhaupt angegangen werden soll und falls ja, welche Anforderungen erfüllt werde sollen und welche nicht (S. 192). Als Lösung bieten sie den Ansatz von Boehm und Huang (zitiert nach Boehm & Huang, 2003), die sog. Value-Driven Software-Engineering, der sich auf die wirtschaftliche Rechtfertigung der Softwareentwicklung konzentriert. Der Ansatz besagt, dass jede Aufgabe in einem Projekt nur so viel wert wie ihr Ergebnis ist und dieser Wert größer als die Kosten sein muss, was wiederum bedeutet, dass der ROI positiv sein muss (S. 193). Boehm und Huang (2003) berechnen den ROI wie folgt:

ROI = (Nutzen – Kosten) / Kosten.

Bei der Entwicklung einer Software ist es angesichts der Kosten wichtig, möglichst fehlerfrei zu entwickeln, da die nachträgliche Fehlerbehebung zusätzliche Kosten verursacht, die 17-26 % der gesamten Lebenszykluskosten einer Software ausmachen können. Umfangreiches Testen der Software vor der Veröffentlichung kann zwar die späteren Kosten für die Fehlerbehebung reduzieren, verlängert allerdings die Entwicklungszeit vor der Veröffentlichung, was ebenfalls zu einem Werteverlust führt. Daher hat jede Projektleitung abzuwägen, ob der durch die zusätzlich benötigte Entwicklungszeit verursachte oder der durch die Fehlerhaftigkeit bedingte Werteverlust größer ist. (Sneed & Jungmayr, 2011, S. 194 - 195). Es werden mehrere Ansätze empfohlen, um die Fehler bei der Entwicklung einer Software zu reduzieren und zwei von denen, die in der vorliegenden Studie angewendet wurden, sind einerseits die Wiederverwendung bestehender Softwarekomponenten, was im Fall von LexiCode Systemdesigns, Komponenten wie UI-Elemente wie Eingabefelder, Forms oder Tests zur Wissensabfrage sind. Ein weiterer Ansatz ist das Testen dieser Systemkomponenten, das sog. Value-driven Testing, das als Maßnahme stark empfohlen wird, weil umfangreiches Testen in der Entwicklungsphase, sowohl für Modul- als auch für Systemtests, die Kosten für die Fehlerbehebung in der Wartungsphase reduzieren kann, was vor allem daran zu erkennen ist, dass die durchschnittlichen Fehlerbehebungskosten als zeitlicher Aufwand in der Testphase innerhalb der Entwicklung bis zu maximal 15 Tagen dauern kann, während es in der produktiven Phase eines Softwareprodukts je nach der Größe des Betriebes, indem die Software eingesetzt wird, mindestens 13 und im schlimmsten Falle 92 Tage dauern kann (Sneed & Jungmayr, 2011, S. 198):

Abb. 3: Tagesaufwand für die Fehlerbehebung

Ein Bild, das Screenshot, Diagramm, Reihe, Astronomie enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung auf der Basis von Sneed & Jungmayr, 2011, S. 198.

Die These von Sneed und Jungmayr (2011), dass allein das Finden von Fehlern den Aufwand für das Testen in Bezug auf den eingesparten Folgekosten rechtfertigen kann, weist eine Schwachstelle auf, weil es sein kann, dass die Fehlerkosten niedriger als die Testkosten sind, was den Testaufwand nicht rechtfertigt und daher ist es wichtig, zuerst zu wissen, wie der Testnutzen ermittelt wird und wie die Testkosten im Voraus zu berechnen sind (S. 196). Um dies zu erreichen bzw. die Testkosten zu ermitteln, schlagen sie die folgende Formel vor:

Aufwand = AF x (TF x TU/TP + TP x (1 – TA)TE x (MT/TB).

Die wichtigsten Parameter dabei sind die Anzahl der Solltestfälle (TF), die angestrebte Testüberdeckung (TU), die manuelle Testproduktivität (TP), der Grad der Testautomatisierung (TA), die Testbarkeitsmetrik (TB), die mittlere Testbarkeit (MT) der Testskalierungsexponent (TE) und der Justierungsfaktor (AF). Das Ergebnis ermittelt eine geschätzte Anzahl von 700 Personentagen, die nötig wäre, um das System während der Entwicklung und dementsprechend vor der Veröffentlichung zu testen. Darüber hinaus wird es für sinnvoll gehalten, den Test-ROI zu ermitteln. Für den Test-ROI ist die Anzahl der erwarteten Fehler zu berücksichtigen, für die ein Wert von 6 Fehlern pro 1000 Zeilen Code, also insgesamt 600 zu erwartende Fehler angenommen wird (Sneed & Jungmayr, 2011, S. 207). Laut einer weiteren Annahme werden mindestens zwei Drittel der vorhandenen Fehler durch systematische Tests aufgedeckt, was die Fehleranzahl auf 200 reduziert und den größeren Anteil von 400 Fehlern nicht erst der Wartungsphase entfernt werden muss. Die Fehler lassen sich nach Ursprung wie folgt verteilen: 40% Anforderungsfehler, 30 % Entwurfsfehler und 30 % Codefehler. Bei der Behebung dieser Fehler wird mit einem geschätzten Aufwand nach Personentagen gearbeitet, der folgendermaßen abläuft: 2,5 Personentage für Anforderungsfehler, 1,0 Personentag für Entwurfsfehler und 0,5 Personentage für Codefehler (Sneed & Jungmayr, 2011, S. 199). Die Berechnung des geschätzten Aufwands für die jeweilige Fehlerquelle ergibt die Fehlerbehebungskosten, die zusammen mit den Kosten des wahrscheinlichen Schadens als Summe den Einsparungspotenzial ergibt, aus dem die Kosten des Testaufwands (Personentage x Tagessatz) subtrahiert und nochmal durch die Testkosten dividiert werden (Sneed & Jungmayr, 2011, S. 207):

ROI = (Nutzen – Kosten) / Kosten.

3. Forschungsdesign

Um die gewünschten Ergebnisse hinter der vorliegenden Forschung zu erzielen bzw. die Forschungsfragen zu beantworten und die Forschungsthese zu bestätigen, wird ein Case Study durchgeführt, das auf dem in der Softwareentwicklung bekannten A/B-Test basiert. Die bereits erwähnte Anwendung zum Deutschlernen, LexiCode, wurde von einer Person im Umfang der Anforderungen, die später erläutert werden, einmal als eine klassische HC-Lösung und parallel als LC/NC-Lösung umgesetzt. Es wurde nicht darauf abgezielt, eine vollständige, ausgereifte und veröffentlichte Anwendung zu entwickeln, sondern vielmehr wurde es angestrebt, die festgelegten Anforderungen an der Anwendung gemäß den beiden Entwicklungsansätzen so umzusetzen, dass sie in einer grundlegenden Form der angeforderten Funktionalität betriebsbereit sind. Somit hat es auch gereicht, dass jede der Anforderungen, wie z. B. die lokale Navigation über Hyperlinks, in den beiden Instanzen der Anwendung nur einmal umgesetzt wird, damit mit überschaubarem Aufwand festgestellt werden kann, ob sich die Anforderungen in der jeweiligen Entwicklungstechnologie überhaupt umsetzen lassen. Sowohl die HC- als auch die LC/NC-Version von LexiCode wurden von einer, und zwar von der gleichen, Person umgesetzt, da die Anforderungen an LexiCode innerhalb dieser Forschung von der Anzahl und der Schwierigkeit der Umsetzung überschaubar sind und somit sich die Beteiligung von weiteren Entwicklern erübrigt hat. Die Implementierung durch einen einzigen Entwickler hat den Kommunikationsaufwand auf null reduziert, was mehr Zeit zur Umsetzung des Projekts gewonnen hat, weil ein großer Teil der Arbeit in einem Team aus Kommunikation besteht, welche in manchen Fällen nicht unbedingt zu produktiven Ergebnissen führt. Die Zeit, die in den Entwicklungsprozess eingeflossen ist, wurde für diese Forschung nicht gemessen, da es nicht damit nicht gerechnet werden konnte, ob sich manche der Anforderungen überhaupt umsetzen lassen. Die in den nächsten Kapiteln beschriebenen Anforderungen wurden so konzipiert, dass sie eine grundlegende Digitalisierung des Prozesses des Deutschlernens befähigen, aber sich an den Erfahrungs- und Kenntnisstand des Entwicklers richten, damit die Entwicklungszeit sich nicht durch den Erwerb neuer Programmierkompetenzen zu viel verlängert. Das Konzept des A/B-Testens wurde an den Bedingungen der vorliegenden Forschung angepasst. Anstatt die unterschiedlichen Variationen von LexiCode unterschiedlichen Menschen vorzustellen, um dann anhand der Konversionsrate dieser Probanden zu prüfen, welche der beiden Versionen effektiver ist (Siroker & Kommen, 2013, S. 8), werden innerhalb der vorliegenden Forschung die beiden Anwendungsversionen von der gleichen Person, nämlich von dem Entwickler, verglichen und selbstverständlich nicht um die Konversionsrate des Entwicklers zu prüfen, sondern einfach um herauszufinden, ob sich eine leistungsfähigere Anwendung mittels HC oder LC/NC bauen lässt. Da der A/B-Test zu einem zunehmend allgegenwärtigen und kritischen Teil von Onlinegeschäften geworden ist (Siroker & Koomen, 2013, S. 8) und LexiCode mit etwaigem Ausbau das Potenzial hat, kommerziell bei einer hohen Nutzerzahl eingesetzt zu werden, eignet sich der A/B-Test als Forschungsmethodik. Des Weiteren eignet sich das A/B-Testen, insbesondere das Werk von Siroker und Koomen (2013) für die Methodik der vorliegenden Forschung, weil sie in ihrer Herausgabe die Bedeutung grundlegender Komponenten vieler Anwendungen und Websites, wie z. B. Calls to Action, Inhalte, visuelle Medien, Seitennavigationen und Formulare, erklären und zu jeder Komponente Ratschläge liefern, wie diese am besten gebaut und getestet werden kann.

3.1 Anforderungsanalyse

Die Anforderungen, die LexiCode als Anwendung zum Lernen von Deutsch zu erfüllen hat, richten sich nach dem typischen Lernweg von Menschen, die nicht nur die deutsche Sprache, sondern jede für sie fremde Sprache beherrschen wollen. Diese potenziellen Lernenden stellen die wichtigsten Stakeholder von LexiCode nach dem Entwickler dar. Dieser soz. *typischer* Lernweg ist bei allen Lernenden hinsichtlich der Reihenfolge der gelernten Inhalte gleich. Diese gelernten Inhalte teilen sich bei dem Fremdsprachenerwerb nach Niveaus auf, variieren von Anfänger (A1) bis zu Fortgeschrittenen (C1) und schließen Grundlegende Sprachkompetenzen ein, wie einfache Begrüßungen und sich vorstellen als Anfänger bis zum Schreiben komplexer Aufsätze und Studieren von Fachliteratur als Fortgeschrittener. Demnach muss eine Anwendung zum Deutschlernen, in diesem Fall, LexiCode, so konstruiert sein, dass sie ihren Nutzern einen Lernbereich bereitstellt, in dem die Lerninhalte nach Niveau aufgeteilt sind, über Theorieeinheiten zuerst als reine Information verinnerlicht und danach durch entsprechende Praxisaufgaben geübt werden können, die idealerweise auch eine Bewertung auf die Leistung liefern. Da in dem bis heute am meisten genutzten Medium zum Lernen von Fremdsprachen, das Lehrbuch, durch die einzelnen Seiten einfach durchgeblättert werden kann, muss auch LexiCode ermöglichen, dass Benutzer mit wenig Anstrengung die Inhalte durchstöbern können. Die Schreibfelder in jedem Lehrbuch befähigen die Lernenden ihre Notizen, Vermerke und Übungsaufgaben direkt im Lehrbuch zu hinterlassen – eine Eigenschaft, die auch in LexiCode vorhanden sein sollte. Fragebögen und Übungen, bei denen die Antworten in Schreibfeldern innerhalb der Fragezeile eingetragen werden, stellen häufige Aufgabenarten in herkömmlichen, gedruckten Lehrbüchern, die entsprechend von einer Anwendung zum Deutschlernen auch bereitgestellt werden sollen. Die bereits erwähnten Vermerke, die in einem Lehrbuch hinterlassen werden, werden von den Lernenden wiederholt aufgeschlagen, um Gelerntes zu wiederholen oder Übungen nochmal neu zu bearbeiten. Daher müssen diese schriftlichen Informationen auch in LexiCode zu jeder Zeit abrufbar sein.

3.2 Spezifikation der Anforderungen

Die in dem letzten Kapitel aufgezählten Anforderungen lassen sich konkretisieren und in den Unterkategorien der funktionalen, der nicht-funktionalen und der technischen Anforderungen untergliedern. Die funktionalen Anforderungen beschreiben die spezifischen Funktionen und Verhaltensweisen, die die Software erfüllen muss:

Tab. 1: Funktionale Anforderungen



Quelle: Eigene Darstellung

F1 muss sicherstellen, dass die Benutzer von LexiCode über Navigationsmenus von einer Inhaltsseite zu einer anderen navigieren können und bei Bedarf noch zur ursprünglichen Seite zurücknavigieren können. Diese Navigation soll uneingeschränkt möglich sein und optimalerweise über Hyperlinks erfolgen. Unter der Anforderung F2 ist zu verstehen, dass die Anwendung den Nutzern nicht nur ermöglicht, Inhalte statisch lesen zu können, sondern diese auch zu ergänzen und zu erweitern. Diese Anforderung lässt sich mit der nächsten Anforderung, F3, weiter spezifizieren, da F3 leisten soll, dass die Benutzer auf der Anwendung Daten eingeben können. Diese Daten sind Antworten zu den Übungsaufgaben und sollen von LexiCode auch bewertet werden, damit die Benutzer aktiv erfahren können, ob sie die Aufgaben richtig oder falsch bearbeitet haben. Die Datenspeicherung, die mit F4 angefordert wird, soll nur die Speicherung der besagten Antworten zu den Übungsaufgaben ermöglichen. Diese Speicherung soll unabhängig von einem Server erfolgen, weil dies, wie in einem Früheren Kapitel erwähnt, nicht im Rahmen der vorliegenden Case Study ist. Die gespeicherten Daten sollen auch bei späteren Sitzungen bzw. Lernvorgängen noch aufrufbar sein und in unveränderter Form vorliegen. F5 soll die User-Experience bequemer machen, indem die Benutzer haben, den Hintergrund der Anwendung auf einer dunklen Farbe umzuschalten und somit den sog. *Dark Mode* aktivieren. Diese Umschaltung wird per Button erfolgen, der nicht tief in einem Einstellungsmenu verborgen, sondern gleich auf der Benutzeroberfläche zu finden sein wird.

Die nachfolgend beschriebenen nicht-funktionalen Anforderungen beschreiben die Qualität von LexiCode in Bezug auf Benutzerfreundlichkeit und Zweckmäßigkeit:

Tab. 2: Nicht-funktionale Anforderungen

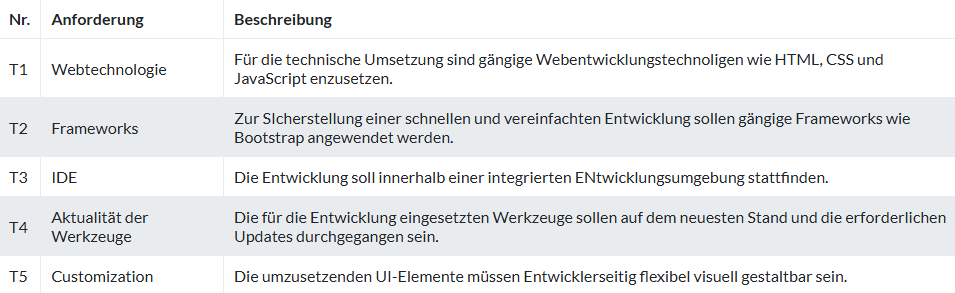


Quelle: Eigene Darstellung

Die Responsivität in Anforderung NF1 soll zum Ergebnis führen, dass die Anwendung, die als eine Desktop-First Anwendung, d. h. primär für die Nutzung mit Desktop-Rechnern und Laptops, entwickelt wird, sich auch an den Bildschirmauflösungen kleinerer mobilen Endgeräte wie Smartphones oder Tablets so anpassen kann, dass die Inhalte noch gut ersichtlich und lesbar sind. Da die gezielte Nutzergruppe aus Menschen bestehen soll, die bereits eine oder mehrere Fremdsprachen gelernt haben oder erst mit LexiCode eine lernen und die App zu privaten Lernzwecken verwenden werden, soll die Anwendung nach NF2 intuitiv zu bedienen sein. Somit soll sich die Anwendung von großen, beruflich eingesetzten Lösungen wie ERP- oder Enterprise Systeme unterscheiden, die oftmals intensive Schulungen voraussetzen, bevor sie produktiv verwendet werden können. NF3 fordert an, dass die Anwendung auf der Benutzeroberfläche eine gute Kombination aus einfachem, nachvollziehbarem Design, geeigneten Farben, passender Schriftgröße und überlegter visueller Hierarchie bietet. In NF4 wird angefordert, dass LexiCode vom Design und von dem Benutzererlebnis her an den herkömmlichen, gedruckten Lehrbüchern erinnert, da dieses Medium meistens von Sprachlernenden bereits eingesetzt worden ist. Somit wird verlangt, dass die Theorieinhalte und Praxisübungen so aussehen und strukturiert sind, wie sie in Lehrbüchern in Papierform zu finden sind. Die letzte, nicht-funktionale Anforderung, NF5, hat den Zweck, Lernenden unabhängig von ihrem Hintergrund zu ermöglichen, mit LexiCode arbeiten zu können. Da Englisch die am meisten gelernte Fremdsprache ist, werden Benutzer mit LexiCode lernen können, da die Inhalte auf Englisch übersetzt sein werden.

Die letzte Gruppe der Anforderungen an LexiCode, die technischen Anforderungen, bezieht sich auf die Technologien, Werkzeuge und Architekturen, die für die Entwicklung der Software verwendet werden:

Tab. 3: Technische Anforderungen



Quelle: Eigene Darstellung

Anforderung T1 wurde festgelegt, weil LexiCode als Webanwendung über einen Browser zugänglich sein soll, ohne dass Nutzer die Anwendung auf ihre Rechner installieren müssen. Diese grundlegende Entwicklung der Anwendung über die in den Anforderungen angegebenen Webentwicklungstechnologien wird auch für zukünftige Projekte ermöglichen, dass LexiCode als eine native Applikation für mobile Endgeräte erweitert werden kann. Über T2 wird verlangt, dass bei der Entwicklung Frameworks eingesetzt werden, weil diese schnelleren Lösungen für gängige Aufgaben in der Entwicklung einer Anwendung bieten. Solche Aufgaben sind z. B. die Einstellung der Textausrichtung, Innen- und Außenabstände der UI-Elemente und UI-Elemente responsiv machen. Die Entwicklung soll zudem, wie in Anforderung T3 angegeben, in einer integrierten Entwicklungsumgebung oder auch *DIE* genannt stattfinden, da diese durch diverse Eigenschaften den Entwicklungsprozess erleichtern und unterstützen. Diese Eigenschaften sind z. B. farbliche Hervorhebung der Code-Schlüsselwörter, Ergänzungsvorschläge und verkürzte Schreibweise von Code-Befehlen. Das Schreiben von Programmcode ist auch über reine Texteditoren wie Notepad oder MS Office Word möglich aber da diese über die als Beispiel aufgezählten Eigenschaften einer *IDE* nicht verfügen, werden die in diesem Case Study für die Entwicklung zumindest nicht eingesetzt. Die eingesetzten Werkzeuge sollen zudem wie in T4 angegeben auf dem aktuellen Stand sein, weil die aktuellen Versionen von Programmiersprachen, Frameworks und *IDE*s alle Standardfeatures weiterhin unterstützen und sogar neue anbieten, die die Entwicklung vereinfachen und die Performance optimieren. Die letzte technische Anforderung, T5, benötigt, dass bei der Entwicklung der Anwendung ausreichend Möglichkeiten zur visuellen und funktionalen Anpassung vorhanden sind.

3.3 Technologie

Die im Rahmen der vorliegenden Forschung eingesetzten Technologien lassen sich in zwei Bereiche unterteilen: die Technologien, die für die Entwicklung der HC-Version von LexiCode herangezogen wurden, und die Technologien, die für die LC/NC-Version verwendet wurden. Um die HC-Version zu verwirklichen, wurden die gängigen Webentwicklungstechnologien HTML, CSS und JavaScript ausgewählt. Die Verwendung von HTML begründet sich dadurch, dass die Markup-Sprache Elemente für viele gängige Anwendungskomponenten bietet wie Formulare, Listen, Texte, Buttons, Überschriften und andere. Zudem können über HTML semantische Elemente wie *<nav>* für Navigation, *<article>* für Artikel oder *<section>* in Anspruch genommen werden, die den Code leichter zu verstehen machen und den gebauten Elementen auch eine semantische Bedeutung zuweisen. Da mittels HTML gestaltete Anwendungen mit CSS ergänzt werden, wurde logischerweise CSS auch in dem vorliegenden Projekt verwendet. CSS bietet vielfältige Anpassungsmöglichkeiten für HTML-Elemente inklusive Breite und Höhe der Elemente, Zeilenabstände der Texte, Schriftgröße, Farben, Schatteneffekte, Animationen und viele andere. Dieses breite Spektrum an Anpassungsmöglichkeiten eignen CSS zur Gestaltung einer ansprechbaren und modernen Benutzeroberfläche. Damit LexiCode auch Funktionen ausführen wurde die Funktionslogik der Anwendung über die Programmiersprache JavaScript geschrieben, die oft in Kombination mit HTML und CSS verwendet wird und aufgrund ihrer modernen ES6 Eigenschaften einen vereinfachten Codesyntax wie Pfeilfunktionen ermöglicht. Des Weiteren lässt sich über JavaScript die bereits angekündigte DOM-Manipulation realisieren, welche aufgrund ihrer Möglichkeiten, die UI-Elemente zu klonen, löschen oder neu zu erstellen, ein mächtiges Werkzeug der Webentwicklung ist und aus diesem Grund JavaScript häufig zu einem bevorzugten Tool bei Webanwendungen macht. Die Entwicklung der HC-Version von LexiCode erfolgt in der Entwicklungsumgebung Visual Studio Code, die wie andere bekannte Entwicklungsumgebungen viele Funktionen zur Verfügung stellt, die gängige Projektverwaltungsaufgaben vereinfachen, wie z. B. das Erstellen von neuen Projektordnern und Dokumenten, die Verknüpfung mit externen Systemen zur Versionsverwaltung wie *Git*, die Ausführung von Terminal Befehlen bei der Arbeit mit Paketmanagern wie *npm* und die Arbeit mit Extensions. Zu den Extensions, die in diesem Projekt angewendet wurden, zählen *Go Live,* der dafür sorgt, dass das aktuelle Projekt nach einem einzelnen Mausklick im Browser ausgeführt wird, *Monokai Pro*, mithilfe deren benutzerdefinierte Farben für die Schlüsselwörter im Programmcode hinterlegt werden können, und *Prettier*, der den Programmcode automatisch formatiert, damit dies der Entwickler nicht tun muss.

Die Programmierung der LC/NC-Version von LexiCode wurde auf der Plattform *Zoho Creator* von dem LC/NC-Anbieter *Zoho* durchgeführt. Diese Plattform wurde für die Low-Code-Anwendung ausgewählt, da sie, laut Angaben auf ihrer Homepage, als die „beste Low-Code-Plattform bewertet“ wird (Zoho, 2025). Zoho Creator bietet die bisher angekündigten Funktionalitäten, die eine typische LC/NC-Plattform kennzeichnen, wie die Möglichkeit, Layouts und Komponenten über Drag-and-Drop zu erzeugen. In der integrierten Entwicklungsumgebung von Zoho Creator, die als *Page Builder* bekannt ist, können einerseits über Drag-and-Drop Diagramme, Suchfelder, Formulare, Berichte, Buttons und andere gängige Komponenten erstellt werden. Andererseits können über Programmcode Anpassungen vorgenommen werden, die über die Drag-and-Drop Bausteine hinausgehen. Dies ist in Bezug auf die Seitenlayouts und -designs über HTML-Snippets möglich und über *Deluge*, die Programmiersprache von Zoho Creator, lässt sich Funktionslogik programmieren. Zudem wird von den Anbietern von Zoho Creator auf deren Homepage angegeben, dass mittels dieser Plattform Anwendungen für jeden Verwendungszweck erstellt, werden können und die Entwicklung mittels Zoho Creator 10-mal schneller in der jeder Phase der Lebenszyklusverwaltung von Apps ist bei anderen Tools (Zoho, 2025). Zoho erklärt diese Geschwindigkeit im Vergleich zu anderen Anbietern mit der Tatsache, dass für eine mittels Zoho Creator gebaute Anwendung kein Setup benötigt, eine Bereitstellung mit einem Klick möglich und kein Wartungsaufwand nötig ist. Die aufgezählten Merkmale einer LC/NC-Plattform, die auch bei Zoho Creator zugänglich sind, zusammen mit der umfassenden Dokumentation, die Tutorials zu der Plattform und die Geschwindigkeit bei der Entwicklung sind die Gründe, warum Zoho Creator eine gute Auswahl für die Entwicklung der LC/NC-Version von LexiCode ist.

3.4 Analyse

Der in der theoretischen Fundierung beschriebene Ansatz des Value-Driven Testings wird auch bei der Entwicklung von LexiCode angewendet. Wie bereits erläutert, ist es bei diesem Ansatz wichtig, den Aufwand, den wahrscheinlichen Schaden und aufgrund dieser Metriken den Test-ROI zu ermitteln. Der Aufwand, der sich über die in der theoretischen Grundlage angegebenen Formel ermitteln lässt, wird in Personentagen gemessen. Der wahrscheinliche Schaden ist die Summe des minimalen und maximalen Schadens geteilt durch 2 (Sneed & Jungmayr, 2011, S. 207):

(Maximaler Schaden + Minimaler Schaden) / 2 = Wahrscheinlicher Schaden

Für den maximalen Schaden rechnen Sneed und Jungmayr (2011) mit den als Beispiel bereits erwähnten 600 Fehlern, die sich als angenommene Fehlerrate von 6 Fehlern pro 100 Zeilen Programmcode äußern. Sie rechnen außerdem mit Sachbearbeitern als Systemnutzern, die auch einen fixen Stundengehalt erhalten, um die Schadenshöhe zu ermitteln. Da allerdings vorgesehen ist, dass LexiCode von privaten Nutzern verwendet wird, werden hier fiktive Daten verwendet, die sich auf private Nutzer beziehen. So werden hier anstatt Sachbearbeiter private Nutzer als die Haupt-User-Gruppe und anstatt Stundengehalt für die Arbeit mit der Anwendung verlorene Arbeitszeit, die ein lernender Nutzer für sein Hauptberuf einsetzen kann, bei dem er oder sie 20 Euro pro Stunde verdient. Wenn zu einem gegebenen Zeitpunkt 100 Nutzer mit LexiCode arbeiten und es zu einem Systemausfall kommt, dann ist es in einem einstündigen Ausfall mit 100 verlorenen Arbeitsstunden zu rechnen und bei einem leichten, halbstündigen Ausfall mit 50. Im schlimmsten Falle, ein maximaler Schaden, führt der Ausfall bei jedem Nutzer zu einem Ausfall von 0,5 Stunden, was 50 verlorene Arbeitsstunden ergibt und im besten Fall sind nur zehn Nutzer betroffen und können eine Stunde mit LexiCode nicht arbeiten, was zu 10 verlorenen Arbeitsstunden führt, was ein minimaler Schaden ist (Sneed & Jungmayr, 2011, S. 207). Bei den angenommenen 600 Fehlern ist bei einer Ausfallquote von 10 % mit mindestens 60 Ausfällen und bei 50 % mit 300 Ausfällen zu rechnen. Der maximale Schaden lässt sich folgenderweise ermitteln:

60 x (100 h a 20 €) = 120.000 €

+ 300 x (50 h a 20 €) = 300.000 €

420.000 €

Der minimale Schaden liegt dann bei:

60 x (50 h a 20 €) = 60.000 €

+ 300 x (10 h a 20 €) = 60.000 €

120.000 €

Für den wahrscheinlichen Schaden ergibt sich somit der folgende Wert:

(420.000 € + 120.000 €) / 2 = 270.000 €

Wenn ein Systemtest mindestens zwei Drittel der existierenden Fehler aufdeckt, kann mit einer Verminderung der Fehleranzahl von 600 auf 200 gerechnet werden, was 400 Fehlern entspricht, die in der Wartung, also in der Betriebsphase nach der Veröffentlichung der Anwendung, nicht behoben werden müssen (Sneed & Jungmayr, 2011, S. 207). Wenn nach den Angaben gerechnet wird, die auch bereits in der theoretischen Fundierung bekanntgemacht wurden, ist für die Behebung der Fehler die folgende Anzahl von Personentagen zu erwarten:

160 Anforderungsfehler x 2,5 = 400 Personentage

120 Entwurfsfehler x 1,0 = 120 Personentage

120 Codefehler x 0,5 = 60 Personentage

Summe: 580 Personentage.

Bei einem Preis von beispielhaften 800 € pro Entwicklerpersonentag liegen die Fehlerbehebungskosten bei 464.000 €. Zusammen mit den Fehlerfolgekosten von 270.000 € ergibt das ein Einsparungspotenzial von 734.000 €.

Ein auf 700 Personentagen geschätzter Testaufwand, bei dem die Kosten pro Tester bei z. B. 720 € pro Tag liegen, berechnen sich die Testkosten wie folgt:

700 x 720 = 504.000 €

Das ROI berechnet sich dann wie folgt:

ROI = (Nutzen – Kosten) / Kosten

ROI = (734.000 – 504.000) / 504.000 = 0,45.

3.5 Fehlerbewertung

Für den Entwicklungsprozess von LexiCode wurden bestimmte Ansätze eingehalten, um die Genauigkeit, Präzision, Wiederholbarkeit und Reproduzierbarkeit von den Ergebnissen zu gewährleisten. Um Richtigkeit sicherzustellen, wurden die Anforderungen an die Anwendung passend überlegt, sodass sie, wie in dem Abschnitt über die Spezifikation der Anforderungen spezifiziert, ein herkömmliches Lehrbuch zum Sprachlernen visuell nachmacht und durch digitale Funktionen ergänzt, digitalisiert und dematerialisiert. Zudem wurden die Anforderungen durch den Entwickler selbst formuliert, der selbst Erfahrung mit dem Erwerb von zwei Fremdsprachen hat und somit eine genauere Vorstellung davon hat, wie der Lernprozess beim Fremdsprachenerwerb abläuft und wie eine digitale Anwendung diesen am besten abbilden kann. Die entwickelten Softwarekomponenten wurden zudem isoliert gleich, nachdem sie entwickelt wurden, getestet, um zeitnah sicherzustellen, dass sie wie geplant funktionieren, ein Ansatz, der vom Prinzip her dem *Test-Driven Development* (*TDD*) ähnelt, nur mit dem Unterschied, dass, anstatt den Testprozess zu automatisieren die Systemkomponenten manuell getestet werden. Bei der Entwicklung wurde auch darauf geachtet, dass bereits angelegte Komponenten wie Buttons oder Container wiederverwendet werden, um in dem Programmcode eine Konsistenz zu sichern.

Es wurde selbstverständlich auch auf Präzision geachtet, damit LexiCode fehlerfrei arbeitet, vor allem wenn es um die Verarbeitung von Daten und Berechnungen geht. Um dies sicherzustellen, wurde zum einen beim Anlegen der Komponenten, in denen vom Benutzer Daten angegeben werden, sichergestellt, dass die Komponenten genau mit dem Datentyp arbeiten können, den der Benutzer angeben wird. Z. B. bei Komponenten auf LexiCode, die Übungsaufgaben darstellen und textuelle Antworten wie Wörter, Satzteile oder ganze Sätze erwarten, wurde im Markup, d. h. im HTML-Code, sichergestellt, dass diese Komponenten mit dem Datentyp „Text“ angelegt werden. Entsprechend wurden Funktionen in der Logik, die die der Richtigkeit von Übungsantworten zu prüfen haben, so geschrieben, dass diese den Datentyp *String* erwarten.

Zum Zweck der Wiederholbarkeit wurde die Versionierung kontrolliert, indem der aktuelle Programmcode nach jeder Änderung wie z. B. eine neugebaute oder geänderte Komponente auf der Web-Plattform *GitHub* gespeichert wird. Die Versionskontrolle über GitHub ermöglicht, dass bei fehlerhaften Entwicklungsarbeiten, die den Programmcode schwer beeinträchtigen und Korrekturen erschweren, die fehlerfreie Version des Codes als Kopie heruntergeladen und an der Anwendung wie geplant weitergearbeitet werden kann. Außerdem hat der Entwickler mittels GitHub die Möglichkeit, von überall und von jedem Gerät auf den Programmcode zuzugreifen und an der Anwendung zu arbeiten. Darüber hinaus wurden die Testszenarien mit den genauen Testparametern dokumentiert, sodass sie unter den gleichen Bedingungen beliebig oft wiederholt werden können.

Damit die Ergebnisse der Teste unter den gleichen Bedingungen wieder durchgeführt werden können, wurden auch Maßnahmen für eine gewisse Reproduzierbarkeit ergriffen. Zum einen wurde die Entwicklungsumgebung dokumentiert, damit sie nachgebildet werden kann. Zum anderen wurden die externen Abhängigkeiten, in diesem Fall das einzige eingesetzte Framework, auch dokumentiert. Es wurden Kriterien für die Erzeugung von unterschiedlichen Testdaten eingesetzt, damit unterschiedliche Testszenarien getestet werden. Auch die Testdaten wurden dokumentiert.

3.6 Technologische Sachzwänge

Die im letzten Kapitel beschriebenen Ansätze zur Genauigkeit und Präzision sowie die Anwendbarkeit und Verfügbarkeit des Projektes weisen gewisse Einschränkungen auf, die zu erwähnen sind, weil sie die Interpretation der Ergebnisse beeinflussen können. Die Einschränkungen liegen größtenteils in der eingesetzten Technologie. Die in dem Abschnitt über die Technologie beschriebenen Werkzeuge, die für die Entwicklung der HC-Version von LexiCode (HTML, CSS und JavaScript) eingesetzt wurden, sind ausführlich dokumentiert und können durch die Lektüre diverser Lernmedien wie Bücher, Blog-Artikel, Zeitschriften, Foren, YouTube Tutorials und Video-Kurse auf Online-Plattformen wie Udemy angeeignet werden. In manchen dieser Quellen sind kann auch die vollständige Umsetzung kleinerer bis mittelgroßer Projekte verfolgt werden, was ein großes Lernpotenzial für unerfahrene Entwickler ist. Da Zoho Creator nicht so lange auf dem Markt wie herkömmliche HC-Entwicklungstechnologien und LC/NC-Entwicklung allgemein eine relativ neue Technologie ist, sind entsprechend weniger Lernmaterialien und Beispielprojekte zu finden, was den Lernprozess verlangsamt und teilweise erschwert. Auch der Entwickler, der sich mit der Entwicklung beider Versionen von LexiCode engagiert hat, kann mehr Erfahrung mit HC- als mit LC/NC-Entwicklung aufweisen. Dies kann dazu geführt haben, dass der Entwickler bei im Zoho Creator gewisse Anforderungen nicht umsetzen konnte, für die es bereits Lösungen oder zumindest Workarounds gibt.

Andererseits kann die Tatsache, dass ein einzelner Entwickler an LexiCode gearbeitet hat, die Richtigkeit negativ beeinflusst haben. In Softwareentwicklungsprojekten, an denen mehrere Entwickler beteiligt sind, können Code Reviews stattfinden, bei denen die Entwickler den Code ihrer Teamkollegen auf Fehler prüfen, damit die Wahrscheinlichkeit von Logikfehlern und Inkonsistenzen verringert wird. Der Aspekt des Erfahrungsmangels bei der Entwicklung der LC/NC-Version von LexiCode kann natürlich auch bei der Entwicklung der HC-Version eine Rolle gespielt haben. In einem Team haben die Entwickler normalerweise unterschiedliche Erfahrungsgrade und die eher erfahrenen können die weniger erfahrenen bei der Entwicklung unterstützen. Auch die reine Höhe des Aufwands kann die Qualität der Arbeit des Entwicklers beeinflusst haben, da Menschen tendieren, Fehler zu begehen, wenn sie von zu viel Arbeit überfordert werden.

Auch der Umfang des Case Studys kann die Aussagekraft und Qualität der Forschung beeinflusst haben. Wie bereits erwähnt, ist für die vorliegende Forschung die Entwicklung einer reinen Frontend-Anwendung geplant worden, ohne dass serverseitige Prozesse, Schnittstellen oder Datenbankanbindungen auch Teil der Anwendung werden, also ohne ein Backend. Eine Anwendung, die über einen Backend verfügt, hat die zweifache Funktionalität einer reinen Frontend-Anwendung und verlangt auch, dass noch mehr Technologien verwendet werden. Dadurch öffnen sich in so einem Case Study weitere Möglichkeiten für einen Vergleich zweier Anwendungsansätze, was die Aussagekraft der Forschung erhöhen kann, weil mehr Anpassungsmöglichkeiten für eine HC-Fullstack-Anwendung im Vergleich zu LC selbstverständlich mehr aussagt als mehr Anpassungsmöglichkeiten für eine Frontend-HC-Anwendung.

Es ist zu erwähnen, dass im Rahmen dieser Studie die kostenfreie Probeversion von Zoho Creator eingesetzt wurde. Obwohl in den kostenpflichtigen Tarifen mehr Features zu finden sind, beziehen sich viele dieser Features auf Merkmale, deren Umsetzung in der vorliegenden Forschung nicht angestrebt wird, wie z. B. KI-Modelle, Integration mit weiteren Zoho Apps oder Business Intelligence und Analytics (Zoho, 2025). Die Features, die in dieser Forschung eigentlich gebraucht werden, wie bspw. Bausteine (Formulare, Berichten, Seiten) und Grundlegende Felder, sind in dem kostenlosen Tarif vorhanden.

4. Ergebnisse und Diskussion

Die Ergebnisse der Entwicklung und somit des Vergleichs der HC- und LC/NC-Version von LexiCode werden in der gleichen Reihenfolge vorgestellt wie die Reihenfolge der Forschungsfrage, die zum Anfang der Studie vorgestellt wurden. Bei der Vorstellung der Ergebnisse werden pro Forschungsfrage, die teilweise auch die Anforderungen nachbilden, im Abwechsel zuerst die Ergebnisse der HC- und danach die Ergebnisse der LC/NC-Anwendung erläutert, bevor es gleichermaßen mit der nächsten Anforderung bzw. mit dem nächsten Vergleichspunkt fortgesetzt wird.

Für die Navigation war es Ziel, sie einerseits lokal für die Inhalte innerhalb eines Sprachniveaus und andererseits global für den Wechsel zu einem anderen Niveau umzusetzen. Die bereits erwähnte und angeforderte Lehrbuchnähe wurde zum Teil implementiert, indem die Lerninhalte innerhalb eines Sprachniveaus wie z. B. C1 visuell als ein Inhaltsverzeichnis angelegt werden, auf dem jeder Punkt im Verzeichnis ein Link zu einer anderen Seite innerhalb der Anwendung ist, auf der sich der entsprechende Inhalt befindet. Diese Struktur wurde für jedes Sprachniveau von A1 bis C1 umgesetzt teilt die Kapitel in Unterkapitel, die die eigentlichen Inhaltsseiten sind. In der HC-Version von LexiCode wurde die lokale Navigation als ein Navigationsmenu anhand des HTML-Elements *<nav>* angelegt, innerhalb dessen eine ungeordnete Liste (*<ul*>) liegt, in der der Link zu jeder Inhaltsseite als Ankerelement (*<a>*) innerhalb eines Listenelementes (*<li>*) eingebettet ist. Über das Attribut *href* wird im Code der Link zu der entsprechenden Seite hinterlegt:

Abb. 4: Link zu einer Inhaltsseite

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Die globale Navigation zur Fortbewegung auf die Inhaltsseiten anderer Sprachniveaus ist Teil der Seitenleistennavigation oder auch *Sidebar-Navigation*. Außerdem bietet die Seitennavigation Links zu allgemeinen Seiten der Anwendung wie z. B. das Kontaktformular oder das appinterne Forum:

Abb. 5: Sidebar-Menu für globale Navigation

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Auch hier sind die Links Ankerelemente, die in den Listenelementen des größeren Navigationselements angelegt wurden. Obwohl auf Zoho Creator auch HTML-Code geschrieben werden kann, werden paar klassische HTML-Attribute, wie eben das *href* Attribut, von der Plattform nicht unterstützt, was in diesem Fall das Anlegen von Hyperlinks und somit das Inhaltsverzeichniskonzept von LexiCode verhindert. Die Seitennavigation konnte zumindest über den App-Menüersteller[[1]](#footnote-1), eine NC-Funktionalität von Zoho Creator, angelegt werden. Diese Funktionalität ermöglicht das Erstellen von Menü- und Untermenüpunkten, indem Abschnitte per Drag-and-Drop in ein Menü reingezogen werden, welches die Seitennavigation darstellt.

Der in der nächsten Forschungsfrage formulierte Vergleichspunkt, der sich auf die Manipulation der Seitenelemente konzentriert, hat für den Umfang dieses Projektes die spezifischere Anforderung, die von den Benutzern angegebenen Daten auf deren Richtigkeit zu prüfen und den Benutzern eine entsprechende Rückmeldung zu liefern. Die Systemkomponente, in der diese Funktionalität erfolgen soll, ist z. B. eine Übung zur Grammatik auf Niveau A1, bei der die Benutzer aufgefordert werden, das richtige Wort wie bspw. eine Begrüßung oder eine Verabschiedung in eine Lücke zu ergänzen. Nach jeder Frage steht der Button „Check“, der die Richtigkeit der Antwort prüft, nachdem er angeklickt wird. Als Beispiel kann die erste Frage in der ersten Übung genannten werden, bei der der Benutzer in das leere Feld vor der Frage „Wie geht es dir?“ eine Begrüßung schreiben muss, die normalerweise morgens geäußert wird („Guten Morgen“). In der HC-Version von LexiCode wurde die Richtigkeitsüberprüfung durch die vorerwähnte DOM-Manipulation realisiert. Dafür wurde für jeden Check-Button eine JavaScript-Funktion geschrieben, bei der das Objekt *document* die Funktion *querySelector()* aufruft, diedafür dient, auf ein HTML-Element entweder nach Klasse oder ID zuzugreifen. In diesem Fall greift *querySelector()* auf die Klasse „check-ex1q1“, weil jede Frage in der Aufgabe, die über HTML als ein *<input>*-Element angelegt wurde, über die Klasse „check“ verfügt, gefolgt von einem Strich, nach dem die Aufgabenbezeichnung mit dem Kürzel *ex#* gefolgt von der Teilaufgabenbezeichnung *q#* steht.

Abb. 6: Input-Feld für Übungsantwort gefolgt von einem Prüf-Button

Ein Bild, das Screenshot, Text, Schrift, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Jede Teilaufgabe wurde in einem einzelnen Paragrafen (*<p>-*Element) platziert, damit jede Teilaufgabe in eine neue Zeile umbricht. In JavaScript greift die *querySelector()-*Funktion ihrerseits auf die Funktion *addEventListener*(), die nach bestimmten Events, also Ereignisse bei der Nutzung der Anwendung lauscht, um dann eine eigens definierte Funktion auszuführen. In den Fall der ersten Übung lauscht unser EventListener auf das *click*-Ereignis, weil die Funktion beim Anklicken des Check-Buttons ausgeführt werden soll. Innerhalb der entwicklerdefinierten Funktion wurde die Variable „ex1q1“ definiert, in der der Zugriff auf den Wert des Eingabefeldes von der ersten Teilaufgabe, ex1q1, gespeichert wird. Variablen anderer Systemkomponenten wie diese wurden auch nach Elementattributen benannt, um eine Entwickler Best-Practise einzuhalten, die besagt, dass Variablen „sprechende Namen“ bekommen sollen. Im weiteren Ablauf der Funktion wurde eine *if*-Abzweigung gebaut, die das erste Teilaufgabenfeld nach dem Inhalt überprüft. Wenn der Inhalt „Guten Morgen“, also richtig ist, wird der Hintergrund des Eingabefeldes mit dem *Hexcode* „#94d82d“ grün und die Buchstaben weiß gefärbt. Wenn die Antwort falsch ist, dann wird der Feldhintergrund mit dem Hexcode „#e03131“ rot und die Buchstaben wieder weiß gefärbt. Die Funktion wurde so geschrieben, dass sie nur noch große Anfangsbuchstaben akzeptiert, weil die Begrüßung ein einzelner Satz ist, bei dem das Wort „Guten“ am Anfang steht. Eine ähnliche Strengheit wurde auch für die anderen Teilaufgaben und Aufgaben umgesetzt.

Die Anforderung nach einem Formular, das über die jeweilige Entwicklungstechnologie auch angepasst werden kann, konnte in der HC-Version von LexiCode gut umgesetzt werden. Dafür wurde im HTML-Code ein *<form>*-Element angelegt, welches standardmäßig für Formulare verwendet wird. Innerhalb des Formulars wurden getrennte *<div>*-Elemente angelegt, die sowohl für kleinere als auch für größere (Teil)Komponenten eingesetzt werden können. In jedem *<div>* befindet sich ein Eingabefeld sowie das dazugehörige Bezeichnungselement, *<label>*. In der bereits erläuterten Übungsaufgabenkomponente wurden Input-Felder verwendet, die bei dem Attribut *type* auf den Wert „text“ gesetzt waren, weil sie Daten in Textformat zu erwarten haben. In dem Formular, das als Kontaktformular dienen soll, wurde neben dem Typ *text* auch ein Input-Feld mit dem Typ *email* angelegt, damit dieses Feld prüfen kann, ob die eingegebenen Daten dem typischen Format von E-Mail Adressen entsprechen:

Abb. 7: Input-Feld für E-Mail-Adressen

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Über CSS können diverse Einstellungen vorgenommen werden, um das Formular visuell anzupassen. Als Beispiel wurde für das Formular eine Außenkante definiert, die 1 Pixel breit ist, fest und blau gefärbt ist. Darüber hinaus wurden die Ecken des Formulars gerundet, es wurden Innenabstände definiert, damit die Felder und deren Bezeichnungen im Formular Abstand zu der Außenkante haben, und es wurden auch Außenabstände definiert, damit das Formular einen gewissen Abstand zu den anderen UI-Elementen hat. Für die Eigenschaft *margin*, die für Außenabstände zuständig ist, wurde als zweiter Wert „auto“ angegeben, weil sich bei drei angegeben Werten für die Eigenschaft *margin* der zweite Wert sich auf die Außenabstände nach links und rechts bezieht. Als Deisgn-Trick wird dieser Wert oft auf auto gesetzt, um eine UI-Komponente zu zentrieren. Die Breite des Formulars wurde auch nach eigenem Wunsch definiert und auf 600 Pixel gesetzt. Um einen 3D-Effekt zu erreichen, wurde dem Formular zudem ein Schatten-Effekt vergeben. Diese CSS-Einstellungen wurden unter der Klasse „*contact-form*“ definiert, welche dem *<form>*-Element im HTML-Code vergeben wurde:

Abb. Eigendefinierte CSS-Klasse für das Kontaktformular

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Das über Zoho Creator angelegte Kontaktformular generierte unterschiedliche Ergebnisse. Obwohl das Formular allgemein optisch ansprechbar ist, waren in dem Form Builder, das Tool innerhalb des Zoho Creators, das für die Erzeugung von Formularen zuständig ist, fast keine Möglichkeiten vorhanden, das Formular optisch anzupassen. Ein Anpassungsversuch wurde z. B. auf die Überschrift des Formulars, „Schreiben Sie uns“, unternommen. Die als eine Anmerkung angelegte Überschrift hat in dem Feld-Editor[[2]](#footnote-2), der wie bei allen anderen Formularfeldern auf der rechten Bildschirmhälfte zu finden ist, die Option, den Text als HTML-Code zu editieren und somit auch die Option, zusätzlichen HTML-Code zu schreiben. Obwohl über den HTML-Editor versucht wurde, die Überschrift mittig zu platzieren, was sowohl in der Vorschau[[3]](#footnote-3), als auch in der finalen Version des Formulars übernommen wird[[4]](#footnote-4). Nachdem der Baukasten ausgewählt wird, in dem alle Formular-Felder vorhanden sind, tauchen auch für diesen Anpassungsoptionen rechts im Form Builder aus, die aber stark eingeschränkt sind. Die durchführbaren Änderungen beziehen sich nur auf den Namen des Abschnitts, in dem das Formular enthalten ist, den Feldlinknamen, die Option, die Kopfzeile für den Abschnitt anzuzeigen und den Feldtyp über ein Auswahlmenü zu ändern. Diese Option ist allerdings nicht änderbar, da das Auswahlmenü ausgegraut ist.

Die Anforderung, in den Übungsaufgaben auf LexiCode Eingabefelder zu bauen, die sich innerhalb des Übungssatzes befinden, die sog. inzeiligen Eingabefelder, kommt wie der Spezifikation der Anforderungen zu entnehmen aus der Idee, die Anwendung an herkömmlichen Lehrbüchern nahzuhalten. Dies wurde in LexiCode zum Teil in dem Abschlusstest zum ersten Kapitel von dem Niveau A1 implementiert. Die zweite Aufgabe, ein Lückentest, hat in jeder Teilaufgabe ein Eingabefeld an einer unterschiedlichen Stelle in dem Satz. Ähnlich zu der Kapitelübersicht im Inhaltsverzeichnis zu jedem Niveau wurde auch hier eine ungeordnete Liste angelegt, in der jede Teilaufgabe ein einzelnes Listenelement ist. In jedem dieser Listenelemente liegen auch hier Input-Felder vom Typ *text*, weil auch hier rein textuelle Daten als Antworten zu ergänzen sind:

Abb. : Inzeilige Eingabefelder mittels HTML

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Das *<ul>*-Element, welches die Listenelemente umschließt, besitzt die vom Entwickler definierte Klasse „bullet-free“, welche defiiniert wurde, um die Stichpunkte zu entfernen, die Listenelemente in einer untergeordneten Liste in HTML automatisch erhalten:

Abb. : Eigene Klasse zum Entfernen der Stichpunkte von Listenelementen

Ein Bild, das Schrift, Text, Grafiken, Typografie enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Da HTML ermöglicht, vor und nach Elementen-Tags Text zu verfassen, konnten je nach Teilaufgabe die Satzteile sowohl vor als auch nach den Input-Feldern geschrieben werden, damit sie wie gewünscht auf der UI auftauchen:

Abb. : HC-Lückentest

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Die Umsetzung der LC/NC-Version hat nahezu das gleiche Ergebnis geliefert. Der HTML- und CSS Code, der für den Abschlusstests für den ersten Kapitel des Niveaus A! geschrieben wurde, wurde ohne Änderung in einem leeren HTML-Snippet im Zoho Creator übernommen. Das Endprodukt ist vom Aussehen her nahezu identisch mit der HC-Version der Augabe[[5]](#footnote-5). Da im Unterschied zu Entwicklungsumgebungen für HC-Entwicklung wie bspw. VSC in einem HTML-Snippet in dem Zoho Creator keine Dokumente oder Ordner angelegt werden können[[6]](#footnote-6), müssen alle für den Abschlusstest relevanten Codezeilen in dem gleichen Snippet eingefügt werden. Dies wird erreicht, indem der CSS-Code in dem HTML-Tag *<style>* eingefügt und der HTML-Code gleich daruntergeschrieben wird[[7]](#footnote-7), was bei zunehmender Anzahl an Komponenten bzw. Codezeilen die Übersichtlichkeit verringern wird.

Die sitzungsunabhängige Speicherung von Daten bzw. Antworten zu den Übungsaufgaben ist eine weitere Anforderung, die für die angesagt Lehrbuchnähe sorgen soll. Die Umsetzung dieser Anforderung in der HC-Version von LexiCode ist wie bei den anderen Komponenten zuerst über HTML und CSS erfolgt. Gebaut wurde ein Lückentest, so wie bei der vorher vorgestellten Komponente mit den inzeiligen Eingabefeldern. Für die aktuelle Komponente wurde jede Teilaufgabe als separates Unterelement in einem Paragrafenelement (*<p>*) angelegt, um sicherzustellen, dass jede Teilaufgabe in einer getrennten Zeile steht. In jedem dieser Paragrafenelemente wurden jeweils zwei Input-Elemente angelegt: das erste als Eingabefeld vom Typ *text*, um die Übungsantworten zu empfangen und das zweite vom Typ *button*, damit die Benutzer die Möglichkeit haben, ihre Antworten manuell zu speichern. Die Buttons haben das Attribut *onclick*, welches eine Art inzeiliges, also „inline“ JavaScript darstellt, und zwar eine Art EventHandler-Funktion, bei der die Verbindung zwischen der EventHandler Funktion und der Komponente, die diese Funktion auslösen soll, anders realisiert wird. Bei diesen Buttons greift nicht die querySelector() Funktion auf die Elemente, sondern das *onclick* bekommt als Wert den Namen der Funktion, also „speichern()“, die beim Betätigen der Buttons ausgeführt werden soll:

Abb. : Teilaufgabe zur sitzungsunabhängigen Datenspeicherung

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Da die EventHandler-Funktion, wie bereits beschrieben, über das onclick-Attribut ausgelöst wird, wird auch die Funktion selbst von der Synrax her anders geschrieben, und zwar als Standard JavaScript Funktion, die mit dem Schlüsselwort *function* definiert wird,gefolgt von dem Funktionsnamen (speichern()). In der Funktion wird jedes Feld als Variable hinterlegt, in der auf das jeweilige Feld durch den querySelector über die Feldklasse auf das Feld zugegriffen wird. Der querySelector greift an sich auf das Objekt *value*, in dem der Wert des jeweiligen Feldes gespeichert wird. Im nächsten Schritt wird über das Objekt *localStorage* die Funktion setItem() aufgerufen, die zwei Parameter übernimmt: den Klassennamen der Komponente und die Variable, in der über den querySelector auf den Wert der jeweiligen (Eingabe)Komponente zugegriffen wird:

Abb. : Funktion zur browserseitigen Speicherung von Daten

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Die Eigenschaft *localStorage* dient der fortbestehenden Speicherung von *Strings*, weil Werte, die über dieses Objekt gespeichert wurden, nach Neuladen der Seite für den Benutzer immer noch verfügbar sind (Flanagan, 2020, S. 537), was die Arbeit mit den Daten in einer späteren Sitzung bzw. Session ermöglicht. Damit die Daten in der gleichen und in einer zukünftigen Sitzung von der *localStorage* Eigenschaft geladen werden, muss dafür eine separate Funktion geschrieben werden. Die dafür geschrieben Funktion „holen()“ legt jedes Eingabefeld aus der gleichen Übung als Variable an, in der *localStorage* de getItem-Funktion auf jedes Feld über sein Klassenname zugreift. In dem nächsten Schritt sorgt eine if-Abzweigung dafür, dass, wenn in einem Feld ein Wert eingegeben worden ist, der Wert in das Feld geladen wird:

Abb. : Holen von zuvor gespeicherten Werten

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Abgesehen von den Anforderungen, auf die sich die Forschungsfragen fokussiert haben, lässt sich der Vergleich zwischen HC und LC/NC auch auf die Ebenen übertragen, die in der Spezifikation der Anforderungen formuliert wurden. Eine dieser Anforderungen, die technischen Anforderungen, verlangt die Verwendung eines Frameworks zur vereinfachten Entwicklung des Applikationsfrontends. Das Framework, das für diesen Zweck gewählt wurde, ist CSS-Bootstrap, weil dieses Framework erlaubt, gängige Komponenteneinstellungen wie Schriftgröße und -ausrichtung, Innen- und Außenabstände, Design von Buttons und ein responsives Design direkt als Klassennamen den Klassen der Komponenten zu vergeben, was damit den Aufwand beim Anlegen eigener entwicklerdefinierten Klassen spart. Zwei Beispiele dafür sind u. a. bei jeder Teilaufgabe in Übung 1 des bereits vorgestellten Abschlusstests zu finden. Jedes Input-Feld besitzt die Bootstrap Klasse *rounded*, welche die Ecken der Komponente, die sie zugewiesen worden ist, rundet:

Abb. : Bootstrap Klasse für runde Ecken



Quelle: Eigene Darstellung

Button-Komponenten können auch mehrere Klassen bekommen. Der „Abgeben“ Button im gleichen Abschlusstest hat z. B. die Klasse *btn*, welche dem Button eine Standardhöhe, -breite und Innenabstand vergibt. Die zweite Bootstrap-Klasse, *btn-primary*, sorgt dafür, dass der Button eine Farbe bekommt, in diesem Fall die blaue Farbe[[8]](#footnote-8), und ein Hover-Effekt, bei dem sich die Farbe auf dunkelblau ändert, wenn der Benutzer mit dem Mauszeiger über den Button schwebt[[9]](#footnote-9):

Abb. : Bootstrap-Klassen für Buttons



Quelle: Eigene Darstellung

Die mit Abstand wichtigste Anpassung, die mittels Bootstrap vorgenommen werden kann, ist, eine Komponente responsiv zu programmieren. Dafür werden zusammenhängende Unterkomponenten oder auch Kindelemente von einem Container- oder auch Elternelement umgegeben. Als Beispiel kann das bereits behandelte Kontaktformular herangezogen werden. Bei der Standardbildschirmauflösung hat das Formular eine Breite von 600 Pixel und befindet sich genau mittig auf dem Display, was über die „margin: auto“ Eigenschaft erreicht wird. Damit der Container des Formulars, das umgebende *<div>*-Element, responsiv wird, wird ihm die Bootstrap Klasse *container* zugewiesen:

Abb. : Responsiver Grid-Container



Quelle: Eigene Darstellung

Bei der Standard Desktop-Browser Bildschirmbreite von über 1700 Pixel steht das Formular, wie beschrieben, mittig auf der Seite, mit viel Abstand zu der linken und zu der rechten Bildschirmkante[[10]](#footnote-10). Wenn die Bildschirmbreite reduziert wird, passt sich der Container des Formulars unabhängig von der Bildschirmbreite so an, dass ein gewisser, wenn auch kleiner, Außenabstand zwischen dem linken Formularrahmen und der linken Browserfensterkante bleibt. Das gleiche gilt für den rechten Rand und die Rechte Browserkante. Bootstrap sorgt auch dafür, dass zwischen jedem Eingabefeld im Formular und den Formularrahmen nach links und rechts Innenabstand vorhanden ist. Im Anhang sind Beispiele für die Responsivität bei Pixel[[11]](#footnote-11) und Pixel[[12]](#footnote-12) zu finden.

5. Fazit

IV. Literaturverzeichnis

Atkinson, R. (1999): *Project management: cost, time and quality, two best guesses and a phenomenon, it‘s time to accept other succes criteria*. Elsevier Science Ltd.

Ajimati, M. O., Carroll, N. & Maher, M. (2025). Adoption of low-code and no-code development: A systematic literature review and future research agenda. The Journal of Systems & Software, 225.

Baumgarten, C., Rainer, E. & Stich, S. (2024): Professionelle Softwareentwicklung mit Low Code optimieren – eine Fallstudie. In: *HMD Praxis der Wirtschaftsinformatik*. Springer Verlag.

Boehm, B. et al. (2000): Software Cost Estimation with COCOMO-II. Prentice Hall.

Brandt-Pook, H. & Kollmeier, R. (2020): *Softwareentwicklung kompakt und verständlich. Wie Softwaresysteme entstehen*. (3. Auflage). Springer Verlag.

Böhler, T. (2023): Software einfacher, flexibler und leichtfüßiger entwickeln. In: *Produktion*. (12):16.

Di Ruscio, D., Kolovos, D. & De Lara, J. (2022): Low-code development and model-driven engineering: two sides of the same coin?

Elshan, E. & Binzer, B. (2024): Mehr als ein Trend?: Wie Low-Code die digitale Transformation unterstützt. In: *HMD Praxis der Wirtschaftsinformatik*. 1070-1087.

Flanagan, D. (2020): *JavaScript: The definitive guide: Master the world’s most-used programming language*. O’Reilly Media, Incorporated

Frank, B. (2023): *ERP:* The next big thing: Sind Lösungen auf Low-Code-Basis die Zukunft? In: *IT-Management*. 50-54.

Gabler Wirtschaftslexikon (2025): *Minimum Viable Product (MVP).* https://wirtschaftslexikon.gabler.de/definition/minimum-viable-product-mvp-119157/version-368108

Gabler Wirtschaftslexikon (2025): Time to Market. https://wirtschaftslexikon.gabler.de/definition/time-market-54271/version-277318

Hensen, U. (2023): Maßgeschnittene Software mittels Low Code. In: *Factory Innovation*. 33-37.

Schreiner, K. (2024): Entwicklungszyklen halbieren mit Low Code. In: *Digital Engineering*. 44-45.

Siroker, D. & Koomen, P. (2013): *A/B Testing. The Most Powerful Way to Turn Clicks into Customers*. Wiley.

Sneed, H. M. & Jungmayr, S. (2011): Mehr Testwirtschaftlichekit durch Value-Driven-Testing. In: Informatik-Spektrum: Organ der Gesellschaft für Informatik e. V. und mit ihr assoziierter Organisationen. Springer-Verlag.

Spierling, D. (2023): Mehr IT-Nachhaltigkeit und Klimaschutz durch Low-Code Development. In: *Wirtschaftsinformatik & Management*. 107–113.

Zoho (2025): *Documentation*. https://www.zoho.com/creator/help/#documentation.

Zoho (2025): *Preisvergleich*. https://www.zoho.com/de/creator/pricing-comparison.html.

Anhang

Anhangsverzeichnis

Anhang A: Bearbeitung der Seitennavigation über den App-Menüersteller

Anhang: HC-Kontaktformular

Anhang: NC-Kontaktformular über Zoho Creator

Anhang: Feldeigenschaften von Formularfeldern

Anhang: HTML-Editor im Form Builder

Anhang: Übungsaufgabe im Zoho Creator

Anhang: HTML- und CSS-Code als getrennte Dokumente

Anhang: HTML-Snippet im Zoho Creator mit HTML- und CSS-Code

Anhang: Bootstrap Button im normalen Zustand

Anhang: Bootstrap Button mit Hover-Effekt

Anhang: Kontaktformular bei Standardbilschirmbreite

Anhang A:

Ein Bild, das Text, Screenshot, parallel, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Anhang:

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Anhang:

Ein Bild, das Screenshot, Text, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

Anhang: Ein Bild, das Text, Screenshot, Software, Computersymbol enthält.

KI-generierte Inhalte können fehlerhaft sein.

Anhang:

Ein Bild, das Text, Screenshot, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

Anhang:

Ein Bild, das Text, Screenshot, Reihe, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Anhang:

Ein Bild, das Text, Screenshot enthält.

KI-generierte Inhalte können fehlerhaft sein.

Anhang:

Ein Bild, das Text, Screenshot, Dokument enthält.

KI-generierte Inhalte können fehlerhaft sein.

Anhang:

Ein Bild, das Text, Screenshot, Schrift, Diagramm enthält.

KI-generierte Inhalte können fehlerhaft sein.

Anhang:

Ein Bild, das Text, Screenshot, Schrift, Diagramm enthält.

KI-generierte Inhalte können fehlerhaft sein.

Anhang:

Ein Bild, das Text, Rechteck, Diagramm, Screenshot enthält.

KI-generierte Inhalte können fehlerhaft sein.

Eidesstattliche Erklärung

Ein Bild, das Text, Schrift, Logo, Grafiken enthält.

KI-generierte Inhalte können fehlerhaft sein.

Hiermit versichere ich an Eides statt, dass ich die Abschlussarbeit selbständig und ohne Inanspruchnahme fremder Hilfe angefertigt habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen. Ich erkläre mich damit einverstanden, dass die Arbeit mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate überprüft wird.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ort, Datum Unterschrift

1. Siehe Anhang [↑](#footnote-ref-1)
2. Siehe Anhang [↑](#footnote-ref-2)
3. Siehe Anhang [↑](#footnote-ref-3)
4. Siehe Anhang [↑](#footnote-ref-4)
5. Siehe Anhang [↑](#footnote-ref-5)
6. Siehe Anhang [↑](#footnote-ref-6)
7. Siehe Anhang [↑](#footnote-ref-7)
8. Siehe Anhang [↑](#footnote-ref-8)
9. Siehe Anhang [↑](#footnote-ref-9)
10. Siehe Anhang [↑](#footnote-ref-10)
11. Siehe Anhang [↑](#footnote-ref-11)
12. Siehe Anhang [↑](#footnote-ref-12)