HAW HAMBURG

BACHELOR THESIS
Ivan Borisovič Bobrov

Documentation-Driven Development mit KI neu denken

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Ivan Borisovič Bobrov

Documentation-Driven Development mit KI neu denken

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Bettina Buth
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 25. März 2025

Ivan Borisovič Bobrov

Thema der Arbeit

Documentation-Driven Development mit KI neu denken

Stichworte

Softwareentwicklung, Softwaredokumentation, Documentation-Driven Development, KI-gestützte Code-Generierung, Prompt Engineering

Kurzzusammenfassung

Dokumentation spielt eine wichtige Rolle in der Softwareentwicklung, doch die Herausforderungen in Bezug auf Aktualität und Konsistenz bleiben häufig bestehen. Documentation-Driven Development versucht, diese Probleme zu lösen, führt jedoch zu einem zusätzlichen Zeitaufwand für Entwickler, die zwischen Dokumentation und Code-Entwicklung jonglieren müssen. Gleichzeitig gewinnen KI-Tools wie GitHub Copilot an Bedeutung, bei denen Prompts oft in natürlicher Sprache erstellt werden und weniger wie traditioneller Code aussehen.

Diese Arbeit untersucht, wie Dokumentation so gestaltet werden kann, dass sie direkt als Kontext für KI-gestützte Code-Generierungstools genutzt werden kann. Das Ziel ist es, Entwicklern zu ermöglichen, sich stärker auf die Entwicklung zu konzentrieren und gleichzeitig eine gute Dokumentation zu gewährleisten, ohne die Produktivität zu beeinträchtigen. Dazu wird eine Methodik entwickelt, die die Dokumentation für die Integration mit solchen Tools optimiert.

Ivan Borisovič Bobrov

Title of Thesis

Rethinking Documentation-Driven Development with AI

Keywords

Software Development, Software Documentation, Documentation-Driven Development, AI-assisted Code Generation, Prompt Engineering

Abstract

Documentation plays a significant role in software development, but challenges regarding its timeliness and consistency often persist. Documentation-Driven Development aims to address these issues but results in additional time demands for developers who must juggle between documentation and code development. At the same time, AI tools like GitHub Copilot are gaining prominence, where prompts are often created in natural language and look less like traditional code.

This work explores how documentation can be designed to serve directly as context for AI-assisted code generation tools. The goal is to enable developers to focus more on development while ensuring effective documentation without compromising productivity. To achieve this, a methodology is developed that optimizes documentation for integration with such tools.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

AI Artificial Intelligence.

BDD Behavior-Driven Development.

CD Continuous Deployment.

CEO Chief Executive Officer.

CI Continuous Integration.

CLI Command-Line Interface.

CSV Comma-Separated Values.

DDD Documentation-Driven Development.

DRY Don't Repeat Yourself.

DSL Domain-Specific Language.

GPT Generative Pre-trained Transformer.

IDE Integrated Development Environment.

IoT Internet of Things.

KI Künstliche Intelligenz.

LLM Large Language Model.

PDF Portable Document Format.

TDD Test-Driven Development.

VS Code Visual Studio Code.

1 Einleitung

Während des Bachelorstudiums stand der Autor vor der Aufgabe, eine Vielzahl unterschiedlicher Technologien zu erlernen. Der Einstieg in jede dieser Technologien erfolgte dabei stets über die entsprechende Dokumentation. Doch nicht jede dieser Lernreisen gestaltete sich angenehm: Einige Dokumentationen waren umständlich geschrieben, unvollständig oder sogar gar nicht vorhanden. Als angehender Informatiker und geübter Leser stellt der Autor sich die Frage, wie man diesen Herausforderungen begegnen kann.

Eine Möglichkeit besteht darin, die Dokumentation als Grundlage für den gesamten Entwicklungsprozess zu nutzen. Durch diesen Ansatz würde der Dokumentation mehr Aufmerksamkeit zuteilwerden, was ihre Qualität und Nützlichkeit erhöhen könnte. Allerdings stellt sich die Frage, ob durch diese Methode nicht auch neue Herausforderungen entstehen oder bestehende Probleme fortbestehen. Wie könnten diese dann überwunden werden?

In dieser Arbeit wird versucht, die oben genannten Fragen zu beantworten. Es wird untersucht, wie eine dokumentationsorientierte Entwicklung, bekannt als Documentation-Driven Development, funktioniert und inwiefern sie die traditionellen Schwachstellen der Software-Dokumentation beheben kann. Zudem wird analysiert, welche neuen Herausforderungen sich durch diesen Ansatz ergeben und wie diese mit Hilfe moderner Technologien, insbesondere künstlicher Intelligenz, überwunden werden können.

Um diese Ziele zu erreichen, wird in Kapitel 2 zunächst erläutert, wie Documentation-Driven Development funktioniert. Dabei wird aufgezeigt, wie dieser Ansatz die herkömmlichen Schwächen der Software-Dokumentation adressiert und welche neuen Herausforderungen möglicherweise entstehen oder bestehen bleiben.

In Kapitel 3 werden die identifizierten Herausforderungen mit den neuesten Fortschritten im Bereich der künstlichen Intelligenz angegangen. Insbesondere wird untersucht, wie Methoden wie Prompt Engineering dazu beitragen können, die Dokumentation effizienter in den Entwicklungsprozess zu integrieren.

Dieses Konzept wird in Kapitel 4 durch einen Proof of Concept veranschaulicht und kritisch evaluiert. Anhand eines praktischen Beispiels wird demonstriert, wie die vorgeschlagene Methodik umgesetzt werden kann und welche Vorteile und Herausforderungen sich daraus für die Softwareentwicklung ergeben.

Diese Arbeit soll als Grundlage für weitere Forschungen in diesem Bereich dienen und dazu beitragen, die Bedeutung von Dokumentation in der Softwareentwicklung neu zu denken.

2 Grundlagen des Documentation-Driven Development

Software-Dokumentation spielt eine wesentliche Rolle im Softwareentwicklungsprozess [Zhi+15]. Sie dient als Kommunikationsmittel zwischen Entwicklern, Testern und anderen Stakeholdern und trägt wesentlich zur Verständlichkeit, Wartbarkeit und Qualität von Softwareprojekten bei [Agh+19; Cho15; DAD06].

Software-Dokumentation lässt sich in verschiedene Typen unterteilen, die jeweils spezifische Aspekte des Entwicklungsprozesses und des Softwareprodukts abdecken. Die folgende Kategorisierung basiert auf den Ausführungen von Rai, Belwal und Gupta [RBG22] und wird in der Abbildung 2.1 veranschaulicht.

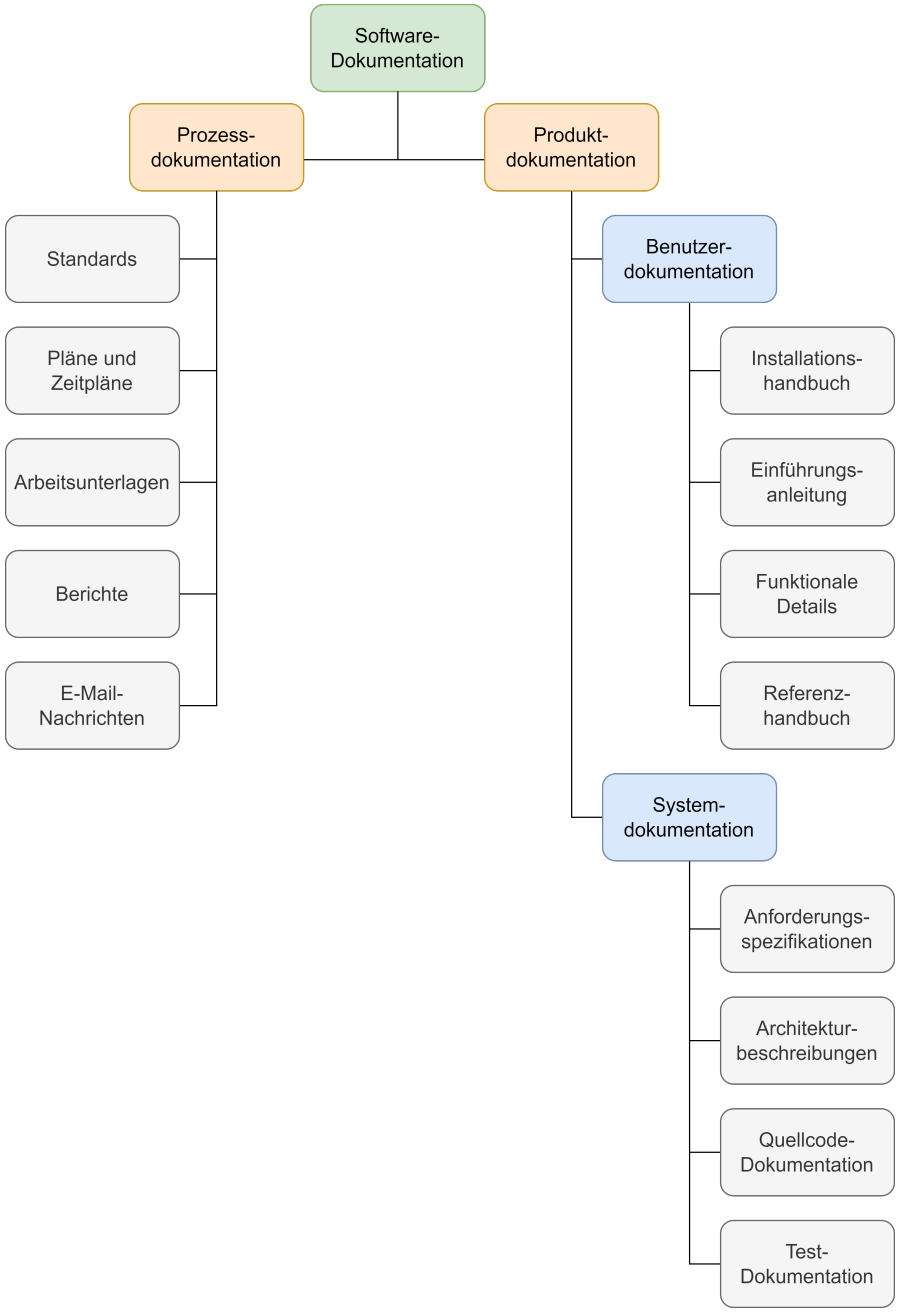

Abbildung 2.1: Typen der Software-Dokumentation nach [RBG22], eigene Darstellung

- **Prozessdokumentation** beschreibt die Abläufe und Verfahren, die während der Softwareentwicklung und -wartung stattfinden. Sie ist wichtig, um den Entwicklungsprozess zu planen, zu überwachen und sicherzustellen, dass er den festgelegten

Standards und Zielen entspricht. Prozessdokumente können in mehrere Unterkategorien unterteilt werden:

  - **Standards** legen fest, wie der Entwicklungsprozess umgesetzt werden soll. Standards können auf organisatorischer, nationaler oder internationaler Ebene erstellt werden.

  - **Pläne und Zeitpläne** werden in der Regel vom Projektmanager erstellt, um den Softwareentwicklungsprozess zu prognostizieren und zu kontrollieren. Pläne können wöchentlich, zweiwöchentlich oder monatlich erstellt werden.

  - **Arbeitsunterlagen** enthalten wesentliche technische Kommunikationen des Projekts, wie Diskussionen der Ingenieure, Zwischenversionen der Produktdokumentation und Implementierungsstrategien.

  - **Berichte** dokumentieren die Ressourcennutzung während des Entwicklungsprozesses.

  - **E-Mail-Nachrichten** dienen als Protokolle der täglichen Kommunikation zwischen den Entwicklungsingenieuren und Managern.

- **Produktdokumentation** bezieht sich auf die Beschreibung des Softwareprodukts selbst, sowohl während der Entwicklung als auch im Betrieb. Diese Dokumente sind besonders nützlich für die Wiederverwendung und Wartung des Systems. Produktdokumentation wird in zwei Hauptunterkategorien unterteilt:

  - **Benutzerdokumentation** richtet sich an verschiedene Benutzertypen und erklärt, wie die Software verwendet werden soll. Benutzerdokumente werden weiter unterteilt in:

    * **Installationshandbuch** beschreibt die Verfahren zur Installation der Software auf verschiedenen Plattformen mit unterschiedlichen Konfigurationen. Systemadministratoren sind die Hauptnutzer dieses Dokuments.

    * **Einführungsanleitung** bietet eine schnelle Beschreibung zur Initialisierung der Software.

    * **Funktionale Details** geben einen Überblick über den Zweck des Systems und beschreiben die erforderlichen Systemdienste.

    - **Referenzhandbuch** erklärt die Systemfunktionen und deren Nutzung und listet Fehlermeldungen sowie entsprechende Wiederherstellungsverfahren auf.

- **Systemdokumentation** enthält die Dokumente, die das Softwareprodukt von den Anforderungen des Kunden bis hin zu den abschließenden Akzeptanztests beschreiben. Sie ist besonders wichtig für die Wartung und Wiederverwendung der Software und ist in der Regel stärker strukturiert als andere Dokumentationstypen:

    - **Anforderungsspezifikationen** dokumentieren die funktionalen und nichtfunktionalen Anforderungen des Kunden.

    - **Architekturbeschreibungen** enthalten die Architektur und Funktionen der Software in Form von Diagrammen, wie z. B. Klassendiagramme, Datenflussdiagramme, Komponentenentwürfe und Datenbankdesigns.

    - **Quellcode-Dokumentation** wird direkt im Quellcode eingebettet. Entwickler nutzen Quellcode-Dokumentation während der Wartungsphase, um den Code besser zu verstehen.

    - **Test-Dokumentation** dokumentiert die Ergebnisse von Systemtests, die auf verschiedenen Ebenen durchgeführt wurden, und hilft Entwicklern dabei, Fehler im System zu beheben.

Trotz ihrer Bedeutung stellt die ordnungsgemäße Pflege der Dokumentation eine erhebliche Herausforderung dar. Häufig ist sie unvollständig, veraltet oder inkonsistent mit dem aktuellen Code, was zu Missverständnissen und erhöhten Wartungskosten führen kann [Ros+13].

Um diese Probleme zu adressieren, wurden verschiedene Ansätze entwickelt. Ein bemerkenswertes Beispiel ist das Konzept des Literate Programming, das von Knuth [Knu99] eingeführt wurde. Er beschreibt Literate Programming als eine Methodik, die eine Programmiersprache mit einer Dokumentationssprache kombiniert, wodurch Programme „robuster, portabler und leichter wartbar“ werden. Die Hauptidee besteht darin, ein Programm als ein Stück Literatur zu behandeln, „adressiert an menschliche Wesen statt an einen Computer“. Diese Herangehensweise betrachtet das Programm als „ein Hypertext-Dokument“ und fördert eine Schreibweise, die sowohl für Menschen als auch für Maschinen verständlich ist [Knu99].

Eine pragmatischere Umsetzung dieses Prinzips findet sich im Documentation-Driven Development (DDD) wieder. DDD legt den Schwerpunkt auf die Erstellung der Dokumentation als ersten Schritt im Entwicklungsprozess. Wie Preston-Werner [Pre10] betont: „Bis Sie über Ihre Software geschrieben haben, haben Sie keine Ahnung, was Sie codieren werden." Procida [Pro16] unterstreicht die Umkehrung der traditionellen Prioritäten von Code und Dokumentation in DDD: „Sie beginnen mit der Dokumentation anstelle Ihres Codes, und anstatt Ihren Code zu dokumentieren, codieren Sie Ihre Dokumentation." Diese Herangehensweise stellt sicher, dass die Dokumentation nicht als nachträglicher Gedanke behandelt wird, sondern als integraler Bestandteil des Entwicklungsprozesses.

Kenlon [Ken17] weist darauf hin, dass es ein verbreitetes Missverständnis darüber gibt, was DDD bedeutet. Oftmals denken Programmierer und Projektmanager, dass der Begriff bedeutet, viele Kommentare im Code zu hinterlassen oder eng mit Dokumentationsautoren zusammenzuarbeiten, während die Entwicklung stattfindet. Er argumentiert jedoch, dass „Ideen nicht einfach vollständig geformt mit jedem Detail in einem ordentlichen Plan auftauchen, der bereit ist, mit einsamen Codezeilen verbunden zu werden. Ideen entstehen allmählich." DDD unterstützt diesen natürlichen Prozess der Ideenentwicklung, indem es den Entwicklern ermöglicht, ihre Gedanken und Konzepte schriftlich zu formulieren, bevor sie sie in Code umsetzen.

Es gibt erfolgreiche Anwendungsfälle in verschiedenen Branchen. O'Driscoll [ODr22] berichtet beispielsweise über die Implementierung von DDD im GOV.UK[1] Sign-In-Projekt, wo die Dokumentation vor dem eigentlichen Coding erstellt wurde, um eine klare Richtung und Effizienz im Entwicklungsprozess zu gewährleisten. Ähnliche Ansätze wurden in Bereichen wie wissenschaftlicher Software [SJK16], der Pharmaindustrie [HN09], eingebetteten Systemen [Zha03] und komplexen Echtzeitsystemen [Luq+04] erfolgreich angewendet. Laut Heeager und Nielsen [HN09] ist DDD besonders relevant für Projekte, die strengen Standards und Zertifizierungen entsprechen müssen. Die Forscher betonen, dass solche Prozessstandards verlangen, „dass Anforderungen vor ihrem Design und ihrer Implementierung spezifiziert werden, dass Dokumentation ein Eckpfeiler zur Erreichung hoher Qualität ist und dass Dokumente nur durch kontrollierte Verfahren geändert werden, also was als dokumentationsgetriebener Ansatz zur Systementwicklung bezeichnet wird". In diesem Kontext stellt die umfassende Dokumentation nicht nur ein Mittel zur Informationsvermittlung dar, sondern ist auch ein zentrales Instrument zur Sicherstellung von Qualität und Compliance [SJK16].

[1] https://www.gov.uk/

Laut Smith, Jegatheesan und Kelly [SJK16] sollte DDD nicht als starres Modell verstanden werden, sondern kann flexibel an unterschiedliche Umgebungen und Anforderungen angepasst werden. Die Forscher betonen, dass aufgrund der vielfältigen Einsatzmöglichkeiten von Softwareprojekten die Implementierungsstrategien von DDD erheblich variieren. Die Methoden und Praktiken, die in einem Kontext erfolgreich sind, erzielen in einem anderen möglicherweise nicht die gleiche Wirkung.

2.1 Philosophie des DDD

Im Zentrum des DDD steht die Priorisierung der Dokumentation vor allen anderen Entwicklungsaktivitäten. Samuel [Sam19] beschreibt dies folgendermaßen: „Die Idee hinter einem solchen Ansatz ist sicherzustellen, dass die Dokumentation für jeden bekannten Aspekt der Anwendung zuerst geschrieben wird und dann das zugehörige Programm implementiert wird. Dies garantiert, dass die Ziele des Programms gut dokumentiert sind.“ Einigermaßen können DDD und Test-Driven Development (TDD) als analoge Ansätze betrachtet werden [Bec03]. Deshalb ist es wichtig zu betonen, dass DDD nicht darauf abzielt, TDD zu ersetzen, sondern vielmehr als Ergänzung zu verstehen ist. Die beiden Ansätze können beispielsweise als „Dokumentieren-Programmieren-Testen-Wiederholen“ zusammengefasst werden [Sam19].

DDD verfolgt mehrere Ziele, die über die reine Dokumentation hinausgehen und einen Mehrwert für das gesamte Entwicklungsteam bieten. Procida [Pro16] und O'Driscoll [ODr22] heben folgende Aspekte hervor:

- **Schaffung eines gemeinsamen, leicht zugänglichen Überblicks auf höherer Ebene über die Arbeit**: DDD ermöglicht es allen Teammitgliedern, ein einheitliches Verständnis des Projekts zu entwickeln.
- **Frühes Hinterfragen des Builds**: Durch die frühzeitige Dokumentation können potenzielle Probleme oder Unklarheiten identifiziert werden, bevor sie in der Implementierung auftreten.
- **Minimierung von Silos oder isolierter Arbeit**: DDD fördert die Zusammenarbeit und den Wissensaustausch innerhalb des Teams.

- **Förderung der Beteiligung von Nicht-Programmierern**: DDD erleichtert die Einbindung von Stakeholdern ohne technischen Hintergrund, indem es die Kommunikation vereinfacht.
- **Einbindung der Programmierarbeit in eine kohärente Erzählung**: Die Dokumentation dient als narrative Struktur, die den Entwicklungsprozess zusammenhängend darstellt.
- **Aufrechterhaltung von Konsistenz**: Eine zentrale Dokumentation hilft dabei, konsistente Praktiken und Standards im gesamten Projekt zu gewährleisten.
- **Bereitstellung einer gemeinsamen, leicht zugänglichen Erfolgsmetrik**: Durch klare Dokumentation können Fortschritte und Zielerreichungen besser gemessen werden.

Damit die Ziele von DDD erreicht werden können, müssen die Dokumente bestimmte interne Qualitätsmerkmale aufweisen. Laut Smith, Jegatheesan und Kelly [SJK16] sollten sie „vollständig, korrekt, konsistent, modifizierbar, nachvollziehbar, eindeutig und verifizierbar“ sein. Ein weiterer wichtiger Aspekt ist die Abstraktion: „Anforderungen sollten abstrakt sein, sodass sie angeben, was erreicht werden soll, aber schweigen darüber, wie es erreicht werden soll.“ Dies fördert ein klares Verständnis der Ziele, ohne die Kreativität und Flexibilität bei der Implementierung einzuschränken.

Die Philosophie hinter DDD basiert auf der Perspektive des Benutzers [Sup14]: „Wenn ein Feature nicht dokumentiert ist, existiert es für den Benutzer nicht, und wenn ein Feature falsch dokumentiert ist, gilt es als fehlerhaft.“ Supalla, Gründer und CEO von Particle[2] - einem Unternehmen, das eine integrierte Internet of Things (IoT)-Plattform als Service anbietet - fasst diese Philosophie in sieben Leitprinzipien zusammen:

1. **Erst dokumentieren, dann entwickeln**: „Wenn es nicht dokumentiert ist, existiert es nicht. Dokumentation ist der beste Weg, ein Feature in den Augen des Benutzers zu definieren“.
2. **Dokumentation vor Entwicklung überprüfen lassen**: Wann immer möglich, sollte die Dokumentation von der Community oder relevanten Stakeholdern überprüft werden, bevor die Entwicklung beginnt.

[2] https://www.particle.io/

3. **Nach Dokumentation (testgetriebene) Entwicklung durchführen**: Die Implementierung folgt der bereits erstellten und validierten Dokumentation, wobei TDD bevorzugt wird.

4. **Mit Unit-Tests Dokumentation testen**: Supalla betont, dass „wenn die Funktionalität jemals nicht mehr mit der Dokumentation übereinstimmt, die Tests fehlschlagen sollten".

5. **Bei Modifikationen zuerst Dokumentation anpassen**: Änderungen an Features sollten immer mit der Aktualisierung der Dokumentation beginnen.

6. **Bei Änderungen an der Dokumentation auch die Tests anpassen**: Die Tests sollten aktualisiert werden, um die Änderungen in der Dokumentation widerzuspiegeln.

7. **Dokumentation und Software synchron halten**: „Dokumentation und Software sollten beide versioniert werden, und die Versionen sollten übereinstimmen, sodass jemand, der mit alten Softwareversionen arbeitet, die entsprechende Dokumentation finden kann".

2.2 Vorgehensweise in DDD

Die Vorgehensweise im DDD kann anhand des Vorschlags von Supalla [Sup14] wie auf der Abbildung 2.2 dargestellt zusammengefasst werden.

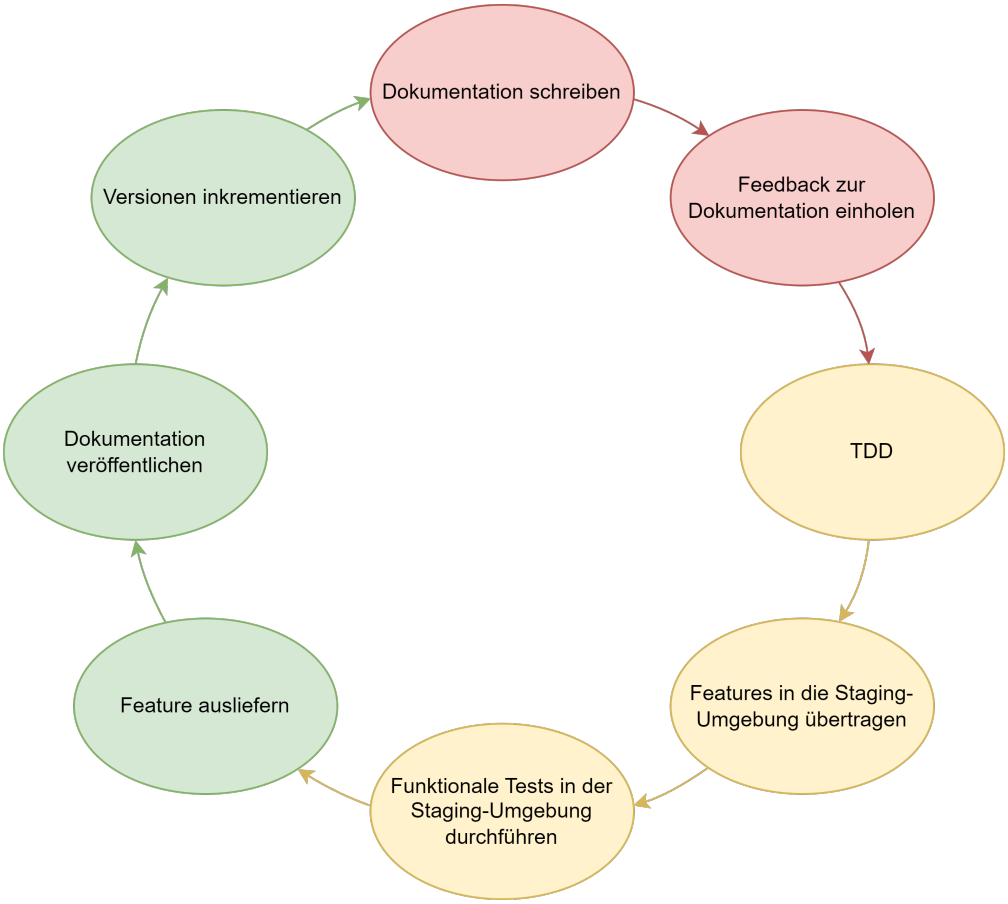

Abbildung 2.2: Vorgehensweise in DDD nach [Sup14], eigene Darstellung

Darauf basierend lassen sich drei zentrale Phasen definieren:

1. Dokumentation schreiben und überprüfen (auf der Abbildung mit rot dargestellt).
2. Nach Dokumentation testen und implementieren (auf der Abbildung mit gelb dargestellt).
3. Feature ausliefern und Dokumentation veröffentlichen (auf der Abbildung mit grün dargestellt).

Es ist wichtig zu unterstreichen, dass DDD nicht zwangsläufig nach dem Wasserfallmodell ablaufen muss [Hee14]. DDD ist flexibel und in verschiedenen Entwicklungsmodellen einsetzbar, einschließlich agiler Methoden [Par19]. Tatsächlich unterstützt eine gut gepflegte Dokumentation den agilen Ansatz, indem sie als lebendiges Dokument dient, das kontinuierlich aktualisiert und an neue Anforderungen angepasst wird [Ken17]. Die einzige

Vorgabe ist, dass die Dokumentation am Ende so erscheinen sollte, als ob ein rationaler Prozess befolgt wurde [SJK16]. Crutchley [Cru22] betont, dass DDD nicht das Prinzip „einmal schreiben und dann fertig“ verfolgt. Laut dem Forscher kann dieses Missverständnis zeitliche oder finanzielle Auswirkungen haben. Es müssen möglicherweise Anpassungen an den Designs vorgenommen werden, bevor die endgültige Veröffentlichung erfolgt. „Dokumente beeinflussen Code, der wiederum die Dokumente beeinflusst.“ Dieser wechselseitige Einfluss zwischen Dokumentation und Code fördert einen dynamischen Entwicklungsprozess, der Anpassungen und Verbesserungen ermöglicht.

2.2.1 Dokumentation schreiben und überprüfen

Eine korrekt erstellte und gepflegte Dokumentation ist von unschätzbarem Wert für den Erfolg eines Softwareprojekts. Preston-Werner [Pre10] betont dies mit den Worten: „Eine perfekte Implementierung der falschen Spezifikation ist wertlos. Nach dem gleichen Prinzip ist eine wunderschön gestaltete Bibliothek ohne Dokumentation ebenfalls nahezu wertlos. Wenn Ihre Software das falsche Problem löst oder niemand herausfinden kann, wie man sie benutzt, läuft etwas sehr schief.“

Das Schreiben der Dokumentation fördert ein tieferes Verständnis des Codes und der zugrunde liegenden Logik. Laut Procida [Pro16], ist dies „der bestmögliche Weg, den Code zu verstehen.“ Indem Entwickler die Funktionalitäten und Abläufe in einer für Menschen verständlichen Sprache beschreiben, identifizieren sie frühzeitig Unklarheiten und können diese vor der Implementierung klären.

Ein weiterer Vorteil des frühen Dokumentierens ist die Verbesserung der Kommunikation innerhalb des Teams. Preston-Werner [Pre10] argumentiert: „Es ist viel einfacher, eine Diskussion auf der Grundlage von etwas Schriftlichem zu führen. Es ist einfach, endlos und im Kreis über ein Problem zu reden, wenn nichts jemals zu Papier gebracht wird. Der einfache Akt, eine vorgeschlagene Lösung aufzuschreiben, bedeutet, dass jeder eine konkrete Idee hat, über die diskutiert und iteriert werden kann.“

Die gemeinsame Dokumentation ermöglicht es zudem, dass Teammitglieder parallel arbeiten können. Wenn alle Beteiligten Zugang zu aktuellen und klaren Dokumentationen haben, können sie unabhängig voneinander an verschiedenen Modulen arbeiten, ohne ständig Rücksprache halten zu müssen [Par19]. Preston-Werner [Pre10] weist darauf hin: „Wenn alle anderen im Team Zugang zu dieser Information haben, bevor Sie das Projekt abgeschlossen haben, können sie mit Zuversicht an anderen Projekten arbeiten, die mit

Ihrem Code interagieren werden. Ohne irgendeine Art von definiertem Interface müssen Sie seriell coden oder riskieren, große Teile des Codes neu zu implementieren.“

Miracle [Mir20] listet mehrere Vorteile auf, die sich ergeben, wenn die Logik in einer menschlichen Sprache vor dem eigentlichen Coding verfasst wird:

- „Es schließt Lücken.“
- „Es verhindert unnötigen Code.“
- „Es verhindert Fehler.“
- „Es macht Tests vollständiger.“

Um den Prozess des Dokumentierens effektiv zu gestalten, empfiehlt Procida [Pro16] einige Schritte:

- „Strukturieren Sie Ihre Dokumentation korrekt.“
- „Machen Sie Ihre Dokumentationsrichtlinien so rigoros wie Ihre Codierungsrichtlinien.“
- „Dokumentieren Sie Ihre Dokumentation.“
- „Wertschätzen Sie Ihre Dokumentationsbeiträge.“

Die Dokumentation kann durch verschiedene Medien bereichert werden, um den Inhalt besser zu vermitteln [Par19]:

- Diagramme visualisieren komplexe Abläufe oder Strukturen und erleichtern das Verständnis.
- Wireframes zeigen Layouts oder Benutzeroberflächen und unterstützen die Gestaltung.
- Videos erläutern Prozesse oder Funktionen in dynamischer Form und sprechen unterschiedliche Lernstile an.

Laut Parsons [Par19] ist Dokumentieren ein kontinuierlicher Prozess, der Disziplin und Konsistenz erfordert und sich selbst nicht steuern kann. Deswegen empfiehlt sie, eine verantwortliche Person oder Rolle zu bestimmen. Dies kann beispielsweise ein Vollzeit-Projektmanager oder ein rotierender Projektleiter sein, der die Verantwortung für die Dokumentation übernimmt.

2.2.2 Nach Dokumentation testen und implementieren

Nachdem die Dokumentation erstellt und überprüft wurde, folgt im DDD der nächste Schritt der Implementierung und des Testens. Die Dokumentation fungiert in diesem Kontext als detaillierter Bauplan für die Anwendung [Ken17]. Kenlon [Ken17] führt weiter aus: „Ihre Anwendungsentwicklung hat sich von der Erstellung von Code, der bei der ersten Überarbeitung verworfen wird, zu einem Gerüst entwickelt, auf dem Sie aufbauen können.“ Durch diese Vorgehensweise wird verhindert, dass unnötiger oder fehlerhafter Code geschrieben wird, der später überarbeitet oder entfernt werden muss. Stattdessen arbeiten die Entwickler von Anfang an zielgerichtet und effizient.

Die Nutzung der Dokumentation als Grundlage für die Implementierung hat auch positive Auswirkungen auf die Zusammenarbeit im Team. Laut Parsons [Par19] kann DDD die Expertise des Teams optimal nutzen. Als Beispiel führt sie an, dass das Benennen von Dingen - eines der herausforderndsten Probleme in der Programmierung - einfach an Dokumentationsautoren delegiert werden kann, die in diesem Bereich im gegebenen Kontext erfahrener sind. So kann jeder die gleiche Sprache verwenden.

In größeren Projekten sind diese Aspekte besonders wichtig. Kenlon [Ken17] erläutert: „Oft stammen die Dokumente von jemandem, der nicht auch der Entwickler ist. Das Ergebnis ist ein fokussierterer Entwicklungszyklus, weil die Entwickler anstelle eines vagen Konzepts einer Anwendung auf einen spezifischen Bauplan hinarbeiten, wie eine Anwendung funktionieren soll. Jedes Menü, jeder Button, jedes Kontextmenü wurde bereits in der imaginären Anwendungsdokumentation abgebildet. Alles, was die Entwickler tun müssen, ist, den Code auszufüllen.“

Das Testen spielt in diesem Schritt ebenfalls eine wichtige Rolle. Indem die Tests auf der Dokumentation basieren, wird sichergestellt, dass der Code die spezifizierten Anforderungen erfüllt [Sup14]. Dadurch können Fehler und Abweichungen frühzeitig erkannt und korrigiert werden. Dies erhöht die Qualität der Software und reduziert den Aufwand für spätere Korrekturen [Mir20].

2.2.3 Feature ausliefern und Dokumentation veröffentlichen

Im letzten Schritt des DDD werden das implementierte Feature ausgeliefert und die dazugehörige Dokumentation veröffentlicht. Dieser Prozess ist entscheidend, um sicher-

zustellen, dass sowohl die Software als auch die Dokumentation synchron und für die Nutzer verfügbar sind [Sup14].

In einer DDD-Umgebung wird die Dokumentation ähnlich wie der Quellcode behandelt. Kenlon [Ken17] betont: „Sie wird in dasselbe Repository wie der Rest des Codes eingecheckt und vor allem anderen aktualisiert.“ Durch dieses Vorgehen wird gewährleistet, dass die Dokumentation stets aktuell ist und Änderungen im Code sofort reflektiert werden.

Laut Procida [Pro16] hat die Veröffentlichung des Features und der Dokumentation auch positive Auswirkungen auf die Softwareentwicklung und die Benutzercommunity. Er betont, dass eine umfassende und zugängliche Dokumentation es neuen Benutzern und Entwicklern erleichtert, das System zu verstehen und sich aktiv daran zu beteiligen. Dies kann zu einer stärkeren Community-Beteiligung und einer schnelleren Weiterentwicklung des Projekts führen.

2.3 DDD als Antwort auf Herausforderungen in der Software-Dokumentation

Die Erstellung und Pflege von Software-Dokumentation bringt zahlreiche Herausforderungen mit sich, die sich auf die Qualität und den Nutzen der Dokumentation auswirken. Aghajani u. a. [Agh+19] identifizierten in ihrer Untersuchung 162 verschiedene Typen von Problemen im Zusammenhang mit Software-Dokumentation, was die Komplexität dieses Themas verdeutlicht. In dieser Bachelorarbeit werden die häufigsten und am meisten diskutierten Herausforderungen in der Literatur untersucht, darunter

- der Umfang und Detaillierungsgrad der Dokumentation,
- die sogenannte Dokumentationsschuld,
- die Kosten-Nutzen-Abwägung sowie
- Fragen der Qualität und Konsistenz.

Diese Auswahl basiert auf den Arbeiten von Satish und Anand [SA16], Wu, Zhou und Wang [WZW21], Muszynski u. a. [Mus+22] und Rost u. a. [Ros+13], die dokumentierten, dass Entwickler sich häufig mit veralteter Dokumentation, mangelnder Rückverfolgbarkeit von Änderungen und dem allgemeinen Desinteresse an Dokumentationsaufgaben

konfrontiert sehen. In den folgenden Abschnitten wird jede Herausforderung im Detail erläutert und diskutiert, inwieweit DDD als Lösungsansatz dienen kann.

2.3.1 Umfang und Detaillierungsgrad der Dokumentation

Die Frage nach dem richtigen Umfang und Detaillierungsgrad der Dokumentation ist nicht trivial. Einerseits kann zu wenig Dokumentation die Verständlichkeit und Wartbarkeit von Software erschweren, insbesondere in agilen Umgebungen, wo minimale Dokumentation oft missverstanden wird und zu Kommunikationsproblemen führen kann [FRS16]. Andererseits neigen viele Systeme dazu, zu viel oder unnötig detaillierte Dokumentationen zu haben, was ebenfalls hinderlich sein kann, da diese den Entwicklern eine große Menge an Informationen präsentiert, die oft nicht relevant für ihre unmittelbaren Aufgaben sind [LSF03].

Plosch, Dautovic und Saft [PDS14] stellten in einer Umfrage fest, dass es eine Tendenz zu „weniger“ Dokumentation gibt, auch wenn keine klare Präferenz erkennbar war. Laut Muszynski u. a. [Mus+22] lässt sich diese Zurückhaltung gegenüber umfangreicher Dokumentation auch in Open-Source-Projekten beobachten, wo die unzureichende und veraltete Dokumentation ein Hindernis für die Akzeptanz der Software darstellt. Die Forscher betonen, dass die Qualität und Abdeckung der Code-Level-Dokumentation oft stark variiert, was potenzielle neue Entwickler davon abhalten kann, zur Software beizutragen.

DDD bietet einen Ansatz, um dieses Problem zu adressieren, indem es die Erstellung notwendiger und relevanter Dokumentation in den Mittelpunkt des Entwicklungsprozesses stellt. Durch das frühzeitige Verfassen von Dokumentation werden Implementierungsdetails vorab geklärt und unnötige Informationen reduziert [Cru22; Pre10]. Laut Crutchley [Cru22] zwingt DDD dazu, „einen Selbst-Feedback-Zyklus über … den Umfang Ihrer Arbeit“ zu durchlaufen. Dies hilft Entwicklern, die Tiefe der Dokumentation kritisch zu bewerten und sicherzustellen, dass nur die notwendigsten Informationen aufgenommen werden.

Da in DDD die Dokumentation aus der Perspektive des Benutzers erstellt wird, hilft dies, eine Überflutung mit unnötigen Informationen zu vermeiden. O’Driscoll [ODr22] berichtet, dass die Benutzer dadurch nicht mit technischen Details überfordert werden, sondern nur die Informationen erhalten, die sie benötigen, um den Service zu nutzen.

Auf der anderen Seite zwingt DDD Entwickler auch dazu, tatsächlich Dokumentation zu erstellen, anstatt sich auf vermeintlich selbstdokumentierenden Code zu verlassen. Miracle [Mir20] gibt zu: „Es hat ewig gedauert, bis ich tatsächlich angefangen habe, meinen Code zu dokumentieren. Ich war immer ein Befürworter davon, dass Code ‚selbstdokumentierend' sein sollte. Aber in Wirklichkeit war dies nur eine reine bequeme Haltung meinerseits. Egal wie gut mein Code geschrieben war, ich konnte ihn nach sechs Monaten beim besten Willen nicht mehr lesen. Wenn ich ihn nach sechs Monaten nicht mehr lesen konnte, wie konnte ich dann von anderen Ingenieuren erwarten, meine interessanten Programmieransätze zu verstehen?"

Durch die Anwendung von DDD wird somit ein Gleichgewicht zwischen zu viel und zu wenig Dokumentation erreicht. Entwickler werden dazu angeleitet, ausreichend Dokumentation zu erstellen, um das Verständnis und die Wartbarkeit der Software zu gewährleisten, ohne dabei in unnötige Details abzuschweifen.

2.3.2 Dokumentationsschuld

Die Dokumentationsschuld ist ein weit verbreitetes Problem in der Softwareentwicklung und tritt auf, wenn Software-Dokumentation nicht mit der Entwicklung des Codes Schritt hält [Mus+22]. Laut Lethbridge, Singer und Forward [LSF03] führt dies zu einer Diskrepanz zwischen der tatsächlichen Software und ihrer Dokumentation, was langfristig die Wartung und Weiterentwicklung eines Systems erheblich erschweren kann. Dokumentationsschuld entsteht insbesondere dann, wenn Änderungen am Code vorgenommen werden, aber die entsprechende Dokumentation nicht zeitnah oder gar nicht aktualisiert wird [Mus+22].

Ein wesentliches Problem der Dokumentationsschuld besteht darin, dass die nicht aktualisierte oder unvollständige Dokumentation zu Verzögerungen führt. Wenn Entwickler veraltete Dokumentation verwenden müssen, um Änderungen oder neue Features zu implementieren, kann dies zu Missverständnissen und Fehlern führen, die den Entwicklungsprozess verlangsamen und die Qualität der Software beeinträchtigen [SUW23]. Diese Verzögerungen können sich in Form von geringerer Wartbarkeit, erhöhtem Aufwand für Nacharbeiten und insgesamt schlechterer Softwarequalität äußern [Mus+22].

DDD bietet einen effektiven Ansatz, um dieses Problem zu adressieren. Da in DDD alle Änderungen mit der Dokumentation beginnen, wird sichergestellt, dass die Dokumentation stets aktuell und konsistent mit dem Code bleibt [Ric14]. Kenlon [Ken17] unterstreicht,

dass Dokumentation - genau wie Coding - kein einmaliger Prozess ist, der nur zu Beginn des Lebenszyklus einer Anwendung stattfindet. Durch die kontinuierliche Aktualisierung der Dokumentation spiegelt sie stets den aktuellen Stand der Software wider, wodurch die Dokumentationsschuld reduziert wird. Dies entspricht der Empfehlung von Gorrod [Gor04, Kapitel 4], Dokumentationen über verschiedene Versionen hinweg konsistent zu halten und Änderungen nachvollziehbar zu machen, um sicherzustellen, dass Tests und Code den aktuellen Anforderungen entsprechen.

Um die Aktualität der Dokumentation zu gewährleisten, schlägt O'Driscoll [ODr22] vor: „Der technische Redakteur könnte mit Nutzerforschern zusammenarbeiten, um die technische Dokumentation regelmäßig zu testen und basierend auf sich ändernden Benutzerbedürfnissen und Feedback zu iterieren." Laut Parsons [Par19] kann die Veraltung der Dokumentation durch die Festlegung von Reviewern - einem Vertreter aus jedem relevanten Team - und Zeitplänen vermieden werden. Sie betont, dass dadurch sichergestellt wird, dass die Dokumentation nicht nur aktuell, sondern auch auf die Bedürfnisse der Benutzer zugeschnitten ist.

Ein praktisches Beispiel für die erfolgreiche Umsetzung von DDD beschreibt O'Driscoll [ODr22]: „Docs-driven Development funktionierte vor dem Start[3] gut, als das Umfeld offen war, aber wir mussten diesen Ansatz bis zum Start und darüber hinaus beibehalten. Um dies zu erreichen, etablierte das Team eine Kultur, die Aktualisierung der Dokumentation als Standard zu betrachten, wenn ein Feature entwickelt oder ein Fehler behoben wird. Ein technischer Redakteur, der sich der Pflege des Documentation-Driven Development widmete, nahm den Dokumentationsaufwand von den Entwicklern. Der technische Redakteur und die Entwickler konnten parallel arbeiten, wobei die Entwickler bauten, während der technische Redakteur sich auf das Schreiben konzentrieren konnte. Das Schreiben oder Aktualisieren der Dokumentation und das Testen der Dokumentation sind Erfolgskriterien für unsere Features. Dies hat dazu beigetragen, einen benutzerorientierten und hinterfragten Build zu erstellen, der die Servicequalität insgesamt verbessert."

Somit kann DDD dazu beitragen, die Dokumentationsschuld zu vermeiden. Durch die explizite Integration der Dokumentation in den Entwicklungsprozess wird sichergestellt, dass die Dokumentation stets aktuell und konsistent mit dem Code bleibt, was langfristig die Wartbarkeit und Qualität der Software verbessert.

[3] Gemeint ist der Start des GOV.UK Sign In.

2.3.3 Kosten-Nutzen-Abwägung

Laut Zhi u. a. [Zhi+15] kann der Aufwand für die Erstellung und Pflege von Dokumentationen erheblich sein, da bis zu 11% der Projektkosten allein auf die Dokumentation entfallen. Die Wissenschaftler weisen darauf hin, dass diese Investition idealerweise durch den Nutzen, den die Dokumentation während der Entwicklungs- und Wartungsphase bietet, gerechtfertigt werden sollte. Laut den Forschern gehören eine verkürzte Bearbeitungszeit von Aufgaben, eine verbesserte Codequalität und eine gesteigerte Produktivität zu den möglichen Vorteilen.

In der Praxis stellt sich jedoch häufig die Frage, ob dieser Nutzen tatsächlich den Aufwand übersteigt. Laut Behutiye u. a. [Beh+20] wird die Dokumentation oft zugunsten der Implementierung vernachlässigt, besonders in agilen Entwicklungsumgebungen, die darauf abzielen, schnell funktionierende Software zu liefern. Die Forscher betonen, dass die Dokumentation unter Zeitdruck häufig als erster Kompromiss gestrichen wird, um Deadlines einzuhalten. Dies führt zu einer unzureichenden Dokumentation, die die langfristige Wartbarkeit und Qualität des Codes beeinträchtigen kann.

Ein weiteres Problem besteht darin, dass der Aufwand für die Erstellung bestimmter Dokumentation oft den erwarteten Nutzen überwiegt [LSF03]. Insbesondere die Erstellung von Architektur-Dokumentation wird als besonders kostenintensiv angesehen [Ros+13]. Dies führt dazu, dass Entwickler selektiv entscheiden, welche Elemente der Software dokumentiert werden sollen, wobei weniger wichtige, aber dennoch nützliche Teile der Software nicht dokumentiert werden [Pan+23].

Ein weiteres Problem ist, dass es keinen umfassenden Ansatz gibt, um die Qualität von Dokumentationen systematisch zu bewerten (siehe Abschnitt 2.3.4). Ohne ein solches Modell ist es laut Zhi u. a. [Zhi+15] schwierig, den tatsächlichen Nutzen der Dokumentation quantitativ zu messen und eine fundierte Entscheidung darüber zu treffen, ob die Kosten gerechtfertigt sind. Obwohl der Aufwand für die Dokumentation zweifellos hoch ist, bleibt die Bewertung des Nutzens in vielen Fällen subjektiv.

DDD hat einen ambivalenten Einfluss auf diese Thematik.

Einige Studien weisen darauf hin, dass DDD den initialen Zeit- und Ressourcenaufwand erhöhen kann. Smith, Jegatheesan und Kelly [SJK16] untersuchten die Anwendung von DDD in der wissenschaftlichen Softwareentwicklung und stellten fest, dass die Teilnehmer den zusätzlichen Zeitaufwand und die erhebliche Vorarbeit als Nachteile empfanden. Es

wurde argumentiert, dass DDD in der Realität nicht funktionieren würde, da oft keine Finanzierung vorhanden ist, um solch detailliert dokumentierte Software zu schreiben. Weitere Kritikpunkte waren, dass DDD eine unnötige Abstraktionsebene darstellt, zu viel Fachjargon aus der Softwaretechnik verwendet und zusätzliche Zeit sowie Ressourcen erfordert, die nicht immer verfügbar sind.

Gleichzeitig bietet DDD jedoch potenzielle langfristige Vorteile. O'Driscoll [ODr22] argumentiert, dass DDD eine kostengünstige und schnelle Methode ist, um Designentscheidungen zu überprüfen, ohne sich vollständig auf deren Umsetzung festzulegen. Durch das frühzeitige Dokumentieren können Teams sicherstellen, dass sie „das Richtige auf die richtige Weise" entwickeln, und dabei Zeit und Geld sparen. Dies reduziert das Risiko von Inkonsistenzen und Fehlern, die später kostspielige Korrekturen erfordern könnten [Ken17]. Zudem stellen Smith, Jegatheesan und Kelly [SJK16] fest, dass DDD zu benutzerfreundlicherem Code und die Rückverfolgbarkeit zwischen Modulen und Anforderungen führt. Die Vorgehensweise bietet laut den Befragten einen systematischeren Ansatz für Architektur und Informationsorganisation, führt zu besser organisiertem Code und erleichtert das Teilen von Informationen über Designentscheidungen.

Die Kosten-Nutzen-Abwägung von DDD ist daher nicht eindeutig. Während der Aufwand für die Dokumentation höher sein kann, können die langfristigen Einsparungen und Qualitätsverbesserungen diesen Mehraufwand rechtfertigen. Smith, Jegatheesan und Kelly [SJK16] weisen darauf hin, dass der zusätzliche Zeitaufwand nicht unbedingt am Anfang des Projekts anfallen muss und dass ein systematischerer Prozess den Bedarf an späteren Refaktorierungen reduzieren kann. Außerdem schlagen sie vor, eine Online-Bibliothek mit fertigen Dokumenten bereitzustellen, um den Zeit- und Arbeitsaufwand zu reduzieren. Es bleibt jedoch eine der größten Herausforderungen, das richtige Gleichgewicht zu finden und sicherzustellen, dass der Nutzen den Aufwand rechtfertigt.

2.3.4 Qualität der Dokumentation

Die Messung der Dokumentationsqualität ist ein Bereich, der in der Forschung verstärkt Aufmerksamkeit erfordert. Zhi u. a. [Zhi+15] stellten fest, dass zwar einige Bemühungen unternommen wurden, die Qualität empirisch zu bewerten, aber es weiterhin an umfassenden Modellen zur systematischen Bewertung mangelt. Insbesondere bei in natürlicher Sprache verfasster Dokumentation ist es schwierig, die Qualität systematisch

zu bewerten, da es an klar definierten Qualitätsmerkmalen mangelt, die zur Bewertung herangezogen werden können [PDS14].

Bei einer Untersuchung zur Qualität von Software-Dokumentation haben Gao, Wang und Wang [GWW23] fünf Dimensionen identifiziert, die Entwickler als besonders wichtig erachten:

- Inhalt,
- Organisation,
- Aktualität,
- Interaktionsdesign und
- Internationalisierung.

Der Inhalt wurde dabei als das wichtigste Kriterium betrachtet, insbesondere für weniger erfahrene Entwickler, die klare Anweisungen und Erklärungen zu grundlegenden Prinzipien erwarten. Darüber hinaus wird eine gut strukturierte Dokumentation geschätzt, die es ermöglicht, benötigte Informationen schnell zu finden. Doch in der Praxis ist dies oft nicht der Fall. Viele Dokumente sind schlecht geschrieben und machen es den Entwicklern schwer, nützliche Informationen zu extrahieren [LSF03].

Eine weitere Herausforderung bei der Sicherstellung der Qualität von Dokumentation ist die Konsistenz. Eine ungenaue oder inkonsistente Dokumentation führt nicht nur zu Verzögerungen im Entwicklungsprozess, sondern erschwert es den Entwicklern auch, Vertrauen in die bereitgestellten Informationen zu entwickeln [LSF03]. Eine umfassende und qualitativ hochwertige Dokumentation sollte jedoch in der Lage sein, Design, Code und Testfälle mit spezifischen Anforderungen zu verknüpfen, um eine vollständige Nachverfolgbarkeit zu gewährleisten [Gor04, Kapitel 4]. Um diese Probleme zu adressieren, schlagen Forscher vor, interne Standards und Terminologien festzulegen sowie Dokumentationsprozesse zu automatisieren, um Konsistenz und Qualität zu gewährleisten [Ros+13].

DDD bietet Ansätze, um diese Qualitätsaspekte zu verbessern.

Ein wesentlicher Vorteil von DDD ist die Verbesserung der Konsistenz der Dokumentation. O'Driscoll [ODr22] betont, dass die Aufrechterhaltung der Teamkonsistenz schwierig

sein kann, insbesondere wenn die Arbeit mehrere Teams umfasst. Durch die Dokumentation, wie ein Produkt oder Service funktioniert, kann diese als einzige verlässliche Informationsquelle dienen.

Darüber hinaus fördert DDD die Kommunikation zwischen den Beteiligten und regelmäßige Überprüfungen, was die Qualität der Dokumentation weiter sichert. O'Driscoll [ODr22] berichtet, dass durch das Teilen des Dokumentationsplans und der technischen Architektur mit dem erweiterten Team „alle ein Verständnis dafür hatten, wie der Build aussehen würde, und zu Iterationen beitragen konnten". Sie betont, dass diese Transparenz ermöglicht es, dass Feedback aus verschiedenen Perspektiven einfließt und die Dokumentation kontinuierlich verbessert wird. Die Überprüfungen sollen jedoch durch eine spezifische Gruppe von Personen durchgeführt werden, anstatt das gesamte Unternehmen zu informieren [Par19].

Ein weiterer positiver Effekt von DDD ist die Motivation der Entwickler zu Beginn der Arbeit. Laut Preston-Werner [Pre10] ist die eigene Begeisterung in diesem Zeitpunkt am höchsten. Diese anfängliche Motivation kann genutzt werden, um mehr Aufmerksamkeit auf die Dokumentation zu lenken und sicherzustellen, dass sie von hoher Qualität ist.

Allerdings garantiert DDD nicht automatisch eine hochqualitative Dokumentation. Parsons [Par19] weist darauf hin, dass es wichtig ist, Maßnahmen zu ergreifen, um die Benutzerorientierung zu stärken. Dazu gehört, auf funktionsfokussierte User Stories zu achten, Benutzer frühzeitig einzubeziehen und kundennahe Teammitglieder wie Support, Community Manager, Developer Advocates und Sales einzubinden.

Einige Entwickler wissen nicht, wo sie anfangen sollen, vergessen, die Dokumentation zuerst zu schreiben, glauben, dass sie es besser visuell erklären können, oder fühlen sich schuldig, nicht mehr Code zu schreiben [Par19]. Um diese Hindernisse zu überwinden, sind klare Richtlinien und Templates hilfreich. Parsons [Par19] schlägt vor, den Prozess zu dokumentieren und Vorlagen mit vorgegebener Struktur und Anleitung zu verwenden, um Entwicklern den Einstieg zu erleichtern und eine konsistente Qualität der Dokumentation sicherzustellen.

Zusammenfassend trägt DDD durch die Förderung von Konsistenz, regelmäßigen Überprüfungen und teamübergreifender Kommunikation erheblich zur Verbesserung der Dokumentationsqualität bei. Obwohl es Herausforderungen gibt, können diese durch den Einsatz von Templates und klaren Richtlinien überwunden werden.

2.3.5 Verbesserungspotenzial

Die bisherigen Ausführungen zeigen, dass DDD als Ansatz zahlreiche Vorteile bietet, um den identifizierten Herausforderungen in der Software-Dokumentation zu begegnen. Es ermöglicht:

- eine bessere Balance zwischen zu viel und zu wenig Dokumentation,
- eine kontinuierliche Aktualisierung, die Dokumentationsschuld reduziert,
- potentielle langfristige Kostenvorteile durch frühzeitige Überprüfung von Designentscheidungen und
- eine Verbesserung der Dokumentationsqualität durch Konsistenz und teamübergreifende Kommunikation.

Dennoch entsteht durch DDD ein erhöhter Zeit- und Ressourcenaufwand für die Erstellung und Pflege der Dokumentation. Diese verbleibende Herausforderung legt nahe, dass DDD von zusätzlichen Technologien und Methoden profitieren könnte, um sein volles Potenzial zu entfalten. Insbesondere der Einsatz Künstlicher Intelligenz könnte einen vielversprechenden Ansatz darstellen, um die Kosten-Nutzen-Abwägung zu verbessern, ohne die oben genannten Vorteile zu gefährden.

3 KI-gestütztes DDD

Laut Hagar und Masuda [HM24] hat die Künstliche Intelligenz (KI) in den letzten Jahren einen tiefgreifenden Einfluss auf verschiedene Bereiche der Gesellschaft, der Technik und der Softwareentwicklung ausgeübt. Die Forscher betonen, dass sie grundlegend verändert, wie wir Technologien nutzen, entwickeln und weiterdenken. Insbesondere in der Softwareentwicklung eröffnet KI neue Möglichkeiten, Prozesse zu optimieren, die Effizienz zu steigern und die Qualität von Softwareprodukten zu verbessern. KI-gestützte Tools unterstützen Entwickler bei einer Vielzahl von Aufgaben, von der Codegenerierung über die Fehlerbehebung bis hin zur automatisierten Codeüberprüfung [OBr+24].

Ein wesentlicher Treiber dieser Entwicklungen sind Large Language Models (LLMs). LLMs wie OpenAIs[4] Generative Pre-trained Transformer (GPT)-Serie und Codex haben bemerkenswerte Fähigkeiten in der Codegenerierung, im Verständnis natürlicher Sprache und in Entwicklerassistenz-Tools gezeigt [Sas+24]. Laut Park u. a. [Par+23] wurden diese Modelle auf umfangreichen Datensätzen trainiert, die aus Webinhalten, Büchern, wissenschaftlichen Artikeln und anderen Textquellen bestehen. Die Forscher weisen darauf hin, dass sie in der Lage sind, menschliche Sprachmuster zu erlernen und komplexe Aufgaben in der natürlichen Sprachverarbeitung zu bewältigen.

White u. a. [Whi+23] beschreiben, dass LLMs in der Softwareentwicklung zunehmend integriert werden, um Entwickler bei der Codeerstellung und -überprüfung zu unterstützen. Laut den Forschern nutzen Tools wie GitHub Copilot[5] LLMs, um Entwicklern Codevorschläge zu unterbreiten, die auf dem Kontext des aktuellen Codes basieren. Udoidiok, Reza und Zhang [URZ24], Dohmke [Doh23], Rodriguez, Dearstyne und Cleland-Huang [RDC23], Coutinho u. a. [Cou+24], Kalliamvakou [Kal22; Kal24], Mariasova u. a. [Mar+24], Daigle und Staff [DS24] und Gao und Research [GR24] weisen darauf hin, dass solche und weitere Anwendungen von KI in der Softwareentwicklung potentielle Vorteile haben:

[4] https://openai.com/
[5] https://github.com/features/copilot

- **Produktivitätssteigerung**: KI-Tools können repetitive Aufgaben automatisieren, wodurch Entwickler mehr Zeit für kreative und komplexe Aufgaben haben.
- **Qualitätsverbesserung**: Durch die Bereitstellung intelligenter Assistenz können KI-Tools dazu beitragen, die Qualität von Softwareprodukten zu erhöhen und Fehler zu reduzieren.
- **Kosteneffizienz**: Die Automatisierung bestimmter Entwicklungsaufgaben kann die Abhängigkeit von teuren menschlichen Ressourcen verringern und somit Kosten einsparen.

Im Rahmen dieser Arbeit stellt sich die Frage, **wie KI-gestützte Werkzeuge in DDD integriert werden können, um die in Abschnitt 2.3.3 beschriebene Kosten-Nutzen-Abwägung zu verbessern**. Dazu ist es zunächst notwendig zu untersuchen, wie eine effektive Interaktion mit Werkzeugen wie GitHub Copilot gestaltet werden kann.

3.1 Interaktion mit KI

Bei der Interaktion mit KI sind die folgenden Aspekte zu berücksichtigen:

1. Prompts.
2. Prompt Engineering.
3. Prompt Patterns.

Laut Mastropaolo u. a. [Mas+23], wird deren Bedeutung besonders im Kontext moderner KI-Tools wie GitHub Copilot deutlich. Sie beschreiben, dass die Softwaretechnik-Forschung sich seit jeher mit der Verbesserung von Codevervollständigungsansätzen befasst hat, die Entwicklern während des Codierens Vorschläge für die nächsten möglichen Token liefern. Mit der Einführung von GitHub Copilot wurde ein bedeutender Schritt nach vorn gemacht. Dieses Tool zeichnet sich insbesondere durch seine beispiellose Fähigkeit aus, sogar ganze Funktionen automatisch aus Beschreibungen in natürlicher Sprache zu generieren. Die Forscher unterstreichen, dass trotz der offensichtlichen Nützlichkeit von Copilot jedoch noch unklar ist, inwieweit es fähig ist, konsistente und korrekte Ausgaben zu liefern, auch wenn sich die Eingaben leicht ändern oder Variationen aufweisen.

Daher ist es für Entwickler entscheidend, ein tiefgreifendes Verständnis für die Interaktion mit solchen KI-Systemen zu entwickeln.

3.1.1 Prompts als Schnittstelle zwischen Mensch und Maschine

In der Interaktion mit LLMs stellen Prompts die zentrale Schnittstelle zwischen Mensch und Maschine dar. Laut White u. a. [Whi+23] ist ein Prompt eine Reihe von Anweisungen oder Eingaben, die einem LLM gegeben werden, um es zu „programmieren“, anzupassen oder seine Fähigkeiten zu verfeinern. Die Forscher betonen, dass durch diese Anweisungen Benutzer das Verhalten des Modells steuern können, indem sie spezifische Regeln und Richtlinien für die Interaktion festlegen. Der Prompt setzt dabei den Kontext für die Konversation und informiert das LLM darüber, welche Informationen wichtig sind und in welcher Form und mit welchem Inhalt die Ausgabe erfolgen soll.

Prompts beeinflussen maßgeblich die nachfolgenden Interaktionen und die vom LLM generierten Ausgaben [Whi+23]. Sie können beispielsweise festlegen, dass ein LLM nur Code erzeugen soll, der einem bestimmten Programmierstil oder Paradigma folgt. Ebenso kann ein Prompt das Modell anweisen, bestimmte Schlüsselwörter oder Phrasen in einem generierten Dokument zu markieren und zusätzliche Informationen zu diesen bereitzustellen [Whi+23]. Durch solche Richtlinien ermöglichen Prompts strukturiertere und nuanciertere Ausgaben, die vielfältige Aufgaben in der Softwareentwicklung unterstützen.

Darüber hinaus können Prompts so gestaltet werden, dass sie ein LLM zu weit mehr befähigen, als nur den Ausgabetyp zu bestimmen oder die bereitgestellten Informationen zu filtern. Laut White u. a. [Whi+23] ist es mit dem richtigen Prompt möglich, völlig neue Interaktionsparadigmen zu schaffen. So kann ein LLM dazu gebracht werden, ein Quiz zu einem bestimmten Softwareentwicklungskonzept oder -werkzeug zu erstellen und durchzuführen oder sogar ein Linux-Terminal zu simulieren. Die Forscher geben das folgende Beispiel:

> *„We are going to play a cybersecurity game. You are going to pretend to be a Linux terminal for a computer that has been compromised by an attacker. When I type in a command, you are going to output the corresponding text that the Linux terminal would produce. I am going to use commands to try and figure out how the system was compromised. The attack should have done one or more of the following things: (1) launched new processes, (2) changed*

> *files, (3) opened new ports to receive communication, (4) created new outbound connections, (5) changed passwords, (6) created new user accounts, or (7) read and stolen information. To start the game, print a scenario of what happened that led to my investigation and make the description have clues that I can use to get started.“*

Zudem beschreiben sie, dass Prompts das Potenzial zur Selbstanpassung besitzen, indem sie weitere Prompts vorschlagen, um zusätzliche Informationen zu sammeln oder verwandte Artefakte zu generieren:

> *„From now on, I would like you to ask me questions to deploy a Python application to AWS. When you have enough information to deploy the application, create a Python script to automate the deployment.“*

Diese fortgeschrittenen Fähigkeiten von Prompts unterstreichen die Bedeutung der Formulierung präziser und effektiver Anweisungen, um einen Mehrwert über einfache Text- oder Codegenerierung hinaus zu bieten.

Im Kontext der KI-gestützten Programmierung wird die Fähigkeit, effektive Prompts zu formulieren, immer wichtiger. Mastropaolo u. a. [Mas+23] heben hervor, dass die Qualität der von KI-Modellen generierten Codevorschläge stark von der Qualität der bereitgestellten Codebeschreibungen abhängt. Entwickler müssen daher lernen, wie sie die gesuchten Codekomponenten präzise beschreiben können, um die Effektivität der KI-Unterstützung zu maximieren.

3.1.2 Grundlagen und Techniken des Prompt Engineerings

Laut [HM24] ist Prompt Engineering ein Konzept innerhalb der KI, das sich mit dem Entwurf und der Verfeinerung von Prompts befasst, um von KI-Modellen spezifische und verbesserte Antworten zu erhalten. Die Forscher merken an, dass es in einigen Kontexten auch als Prompt Design bezeichnet wird. Im Wesentlichen handelt es sich beim Prompt Engineering um die Methode, mit der LLMs über Prompts „programmiert“ werden [Whi+23].

Die Bedeutung des Prompt Engineerings zeigt sich besonders in der Softwareentwicklung, wenn LLMs zur automatischen Codegenerierung eingesetzt werden. Mastropaolo u. a. [Mas+23] betonen, dass wir nicht genau wissen, inwieweit semantisch gleichbleibende Änderungen in der natürlichen Sprachbeschreibung, die einem Modell bereitgestellt

wird, die generierte Codefunktion beeinflussen. Ihre empirische Studie umfasste die automatische Generierung von 892 Java-Methoden mit Hilfe von GitHub Copilot, basierend auf deren ursprünglicher Javadoc-Beschreibung. Anschließend wurden für jede Methode verschiedene semantisch äquivalente Beschreibungen sowohl manuell als auch automatisch erstellt, um zu analysieren, inwieweit sich die von Copilot generierten Codevorschläge änderten. Die Ergebnisse zeigten, dass das Ändern der Beschreibung in etwa 46% der Fälle zu unterschiedlichen Codeempfehlungen führte. Zudem konnten Unterschiede in den semantisch äquivalenten Beschreibungen die Korrektheit des generierten Codes um bis zu ±28% beeinflussen.

Laut Mastropaolo u. a. [Mas+23] unterstreichen diese Erkenntnisse die zentrale Rolle des Prompt Engineerings in der Interaktion mit KI-Systemen. Die Forscher betonen, dass es entscheidend ist, dass Entwickler lernen, effektive und präzise Prompts zu erstellen, da verschiedene Formulierungen zu unterschiedlichen Ergebnissen führen können. Dies gewährleistet nicht nur konsistente und zuverlässige Ergebnisse, sondern trägt auch zur Verbesserung der Nutzbarkeit von KI-gestützten Codeempfehlungssystemen bei.

Trotz der zahlreichen Techniken und Strategien im Prompt Engineering [Hou+24] stehen Entwickler vor erheblichen Herausforderungen, die eine effektive Nutzung von LLMs in der Softwareentwicklung erschweren.

Laut Rodriguez, Dearstyne und Cleland-Huang [RDC23] besteht eine der größten Herausforderungen darin, dass kleine Änderungen an Prompts zu signifikanten Unterschieden in den Modellausgaben führen können. Die Forscher weisen darauf hin, dass subtile Anpassungen wie das Pluralisieren von Wörtern, das Austauschen von Präpositionen oder das Umstellen von Phrasen die Ergebnisse deutlich beeinflussen können. Außerdem identifizieren LLMs häufig andere Beziehungen zwischen Artefakten als menschliche Experten, was dazu führt, dass die erzeugten Ausgaben nicht immer mit den gewünschten Zielen übereinstimmen. Um diese Diskrepanzen zu minimieren, schlagen die Forscher vor, dass Prompts den spezifischen Verwendungszweck der erzeugten Ergebnisse klar angeben. Ihrer Meinung nach kann dies dazu beitragen, die Ausgabe des Modells besser auf das gewünschte Ergebnis abzustimmen und eröffnet gleichzeitig die Möglichkeit, verschiedene Ansichten zu erstellen - ein potenzieller Vorteil gegenüber rein ähnlichkeitbasierten Methoden.

Darüber hinaus beschreiben Rodriguez, Dearstyne und Cleland-Huang [RDC23], dass die optimale Prompting-Strategie von verschiedenen Faktoren abhängt, darunter die verfügbaren Ressourcen, das verwendete Modell und das angestrebte Nutzungsszenario. Unter-

schiedliche LLMs weisen spezifische Stärken und Schwächen auf und können unterschiedliche Prompts benötigen, um auf denselben Datensätzen die gewünschten Ergebnisse zu erzielen. Die Forscher merken an, dass selbst Variationen zwischen Versionen desselben Basismodells die Leistung bei derselben Aufgabe beeinflussen können. Diese Variabilität erschwert die Standardisierung von Prompting-Methoden und erfordert ein tiefgreifendes Verständnis der Modelleigenschaften.

Angesichts dieser Herausforderungen ist klar, dass weitere Forschung im Bereich des Prompt Engineerings notwendig ist. Laut Sasaki u. a. [Sas+24] ist der Bereich noch fragmentiert und es fehlt an einem umfassenden Verständnis. Die Forscher unterstreichen, dass obwohl die Anwendung dieser Ansätze rasant zunimmt, gibt es noch keine etablierten Best Practices oder standardisierten Methoden, um konsistente und zuverlässige Ergebnisse zu erzielen.

3.1.3 Muster im Prompt Engineering

Prompt Patterns sind essenzielle Komponenten des Prompt Engineerings und dienen als wiederverwendbare Lösungen für spezifische Probleme bei der Interaktion mit LLMs [Whi+23]. Sie ähneln Software-Entwurfsmustern, konzentrieren sich jedoch speziell auf die Generierung von Ausgaben durch LLMs wie ChatGPT. Durch die Anwendung von Prompt Patterns können Benutzer die Ausgabe und Interaktion mit LLMs gezielt anpassen und optimieren.

Ein gutes Beispiel für ein Prompt Pattern ist „Kontextmanager“, wie es in der Arbeit von White u. a. [Whi+23] beschrieben wurde. Die Forscher definieren die Idee dieses Patterns als eine Methode, die es Nutzern ermöglicht, den Kontext einer Konversation mit einem LLM explizit anzugeben oder zu entfernen. Ziel dieses Patterns ist es, den Fokus der Interaktion auf spezifische Themen zu legen oder irrelevante Themen auszuschließen. Dadurch erhalten Benutzer eine stärkere Kontrolle darüber, welche Informationen das LLM bei der Generierung von Ausgaben berücksichtigen oder ignorieren soll.

White u. a. [Whi+23] haben die Schlüsselelemente definiert, die ein Prompt für dieses Pattern enthalten sollte:

- *„Within scope X“*
- *„Please consider Y“*

- *„Please ignore Z“*
- *„(Optional) start over“*

Für einen solchen Prompt geben die Forscher folgendes Beispiel:

> *„When analyzing the following pieces of code, only consider security aspects.“*

Damit wird das LLM angewiesen, sich ausschließlich auf sicherheitsrelevante Informationen zu konzentrieren, während andere Aspekte, wie etwa Formatierung oder Namenskonventionen, bewusst ausgeblendet werden.

Die Dokumentation und Nutzung von Prompt Patterns im Kontext der Automatisierung von Softwareentwicklungsaufgaben ermöglicht es Einzelpersonen und Teams, Einschränkungen für die generierte Ausgabe durchzusetzen, sicherzustellen, dass relevante Informationen enthalten sind, und das Interaktionsformat mit dem LLM anzupassen, um die in Abschnitt 3.1.2 beschriebenen Probleme effizienter zu lösen [Whi+23]. Prompt Patterns können als Pendant zu allgemeinen Software-Entwurfsmustern betrachtet werden, jedoch angepasst an den spezifischen Kontext der LLM-Ausgabeerzeugung.

Laut White u. a. [Whi+23] folgt ein typisches Prompt Pattern einem strukturierten Format, ähnlich dem von klassischen Software-Entwurfsmustern, mit Anpassungen für die spezifischen Anforderungen der LLM-Ausgabe. Dieses Format beinhaltet:

- **Name und Klassifikation**: Ein eindeutiger Name, der das adressierte Problem widerspiegelt, und eine Kategorisierung des Musters, beispielsweise in Bereiche wie Ausgabekonfiguration oder Fehlererkennung.
- **Absicht und Kontext**: Beschreibung des Problems, das das Muster löst, und der Ziele, die es erreicht. Das Problem sollte idealerweise domänenunabhängig sein.
- **Motivation**: Begründung der Wichtigkeit des Problems und wie das Muster die Interaktion mit dem LLM verbessert.
- **Struktur und Schlüsselideen**: Darstellung der grundlegenden Informationen, die das Muster dem LLM bereitstellt, ähnlich den „Teilnehmern“ in Software-Entwurfsmustern.
- **Beispielimplementierung**: Konkrete Formulierung des Musters in der Praxis.
- **Konsequenzen**: Zusammenfassung der Vor- und Nachteile der Anwendung des Musters und Hinweise zur Anpassung an verschiedene Kontexte.

Es gibt verschiedene Versuche, Prompt Patterns in der Softwareentwicklung zu klassifizieren und zu strukturieren. Ein Ansatz von White u. a. [Whi+23] kategorisiert Prompt Patterns wie in der Tabelle 3.1 dargestellt:

Tabelle 3.1: Klassifikation von Prompt Patterns im Kontext der Softwareentwicklung

Kategorie	**Prompt Pattern**
Input-Semantik	Erstellung einer Metasprache
Ausgabekonfiguration	Ausgabeautomatisierung
	Persona
	Visualisierungsgenerator
	Rezept
	Vorlage
Fehlererkennung	Faktencheckliste
	Reflexion
Prompt-Verbesserung	Fragenverfeinerung
	Alternative Ansätze
	Kognitiver Prüfer
	Ablehnungsunterbrecher
Interaktion	Umgedrehte Interaktion
	Spielablauf
	Unendliche Generation
Kontextsteuerung	Kontextmanager

- **Input-Semantik**: Behandelt, wie ein LLM die Eingabe versteht und in nutzbare Informationen für die Ausgabe umsetzt.
- **Ausgabekonfiguration**: Fokussiert auf die Anpassung von Typ, Format, Struktur oder anderen Eigenschaften der generierten Ausgabe.
- **Fehlererkennung**: Zielt darauf ab, Fehler in der generierten Ausgabe zu identifizieren und zu beheben.
- **Prompt-Verbesserung**: Konzentriert sich auf die Verbesserung der Qualität von Eingabe und Ausgabe.
- **Interaktion**: Behandelt die Dynamik zwischen Benutzer und LLM.
- **Kontextsteuerung**: Steuert die kontextuellen Informationen, innerhalb derer das LLM operiert.

In einer anderen Studie identifizierten Sasaki u. a. [Sas+24] 21 Prompt-Engineering-Muster und ordneten sie in fünf Hauptkategorien ein:

- **Lernansatz**: Muster, die den notwendigen Kontext für LLMs bereitstellen, einschließlich Few-Shot- und Zero-Shot-Learning.
- **Interaktionsfokussierte Methoden**: Techniken zur Bereicherung der Ausgabe, von einfachen Anfragen bis zur umfangreichen Aufgabenzerlegung durch wechselseitige Interaktion.
- **Aufgabenspezifische Muster**: Spezifische Muster für softwaretechnische Aufgaben wie Codegenerierung oder Kommentarerstellung.
- **Modelloptimierung**: Muster zur Feinabstimmung der Modellparameter für bestimmte Aufgaben.
- **Systematisierung und Katalogisierung**: Sammlung und Strukturierung von Prompt Patterns für Softwareentwicklungsaufgaben.

Diese unterschiedlichen Klassifikationen zeigen, dass es keine einheitliche Standardisierung von Prompt Patterns gibt. Dennoch bleibt die zentrale Idee bestehen, Best Practices in der Interaktion mit LLMs zu extrahieren und wiederzuverwenden, um die Effektivität in der Softwareentwicklung zu steigern.

Die Notwendigkeit weiterer Forschung in diesem Bereich ist laut White u. a. [Whi+23] offensichtlich. Die Wissenschaftlicher betonen, dass die Dokumentation der Prompt Patterns nicht ausreichend ist, obwohl sie als Mustersammlung nützlich ist. Es besteht ein Bedarf an kontinuierlicher Arbeit zur Verfeinerung und Erweiterung bestehender Muster sowie zur Erforschung neuer und innovativer Anwendungsmöglichkeiten von LLMs. Da sich die Fähigkeiten von LLMs weiterentwickeln, schlagen die Forscher vor, die Muster regelmäßig zu überprüfen und anzupassen, um ihre Relevanz und Funktionalität zu gewährleisten.

3.2 Integration von DDD und KI-gestützter Codegenerierung

In DDD wird die Software-Dokumentation vor der eigentlichen Implementierung erstellt, was einen deutlichen Unterschied zu anderen Entwicklungsansätzen darstellt (siehe Ab-

schnitt 2.2). Diese Herangehensweise stellt jedoch eine Herausforderung hinsichtlich der Kosten-Nutzen-Abwägung dar, da der initiale Aufwand für die Erstellung ausführlicher Dokumentation hoch sein kann (siehe Abschnitt 2.3.3).

Hier bietet die KI-gestützte Codegenerierung eine vielversprechende Lösung. Laut einer Umfrage von Stack Overflow [Sta24] sehen die meisten Entwickler die Codegenerierung als eine der häufigsten Anwendungen von KI-Tools (siehe Abbildung 3.1).

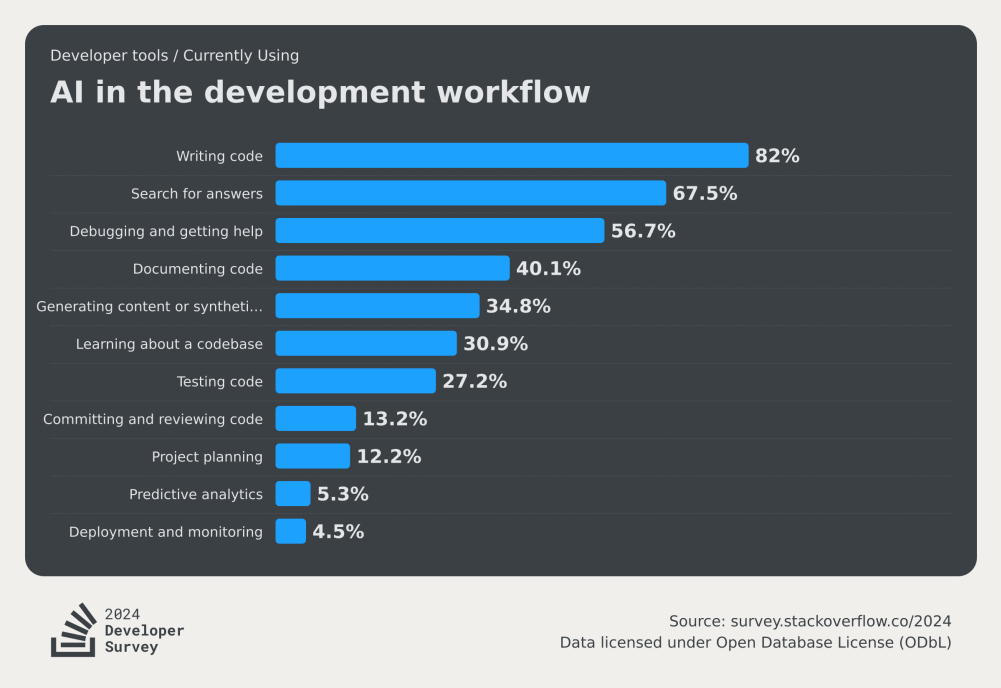

Abbildung 3.1: Häufigste Anwendungsfälle für KI-Tools in der Softwareentwicklung [Sta24]

Majdinasab u. a. [Maj+24] beschreiben, dass Codegenerierungswerkzeuge darauf abzielen, die Produktivität zu steigern, indem sie Codeabschnitte für Entwickler generieren - entweder durch Autovervollständigung bestehender Codefragmente oder durch die Umwandlung von Prompts in natürlicher Sprache in Code. Die Verwendung von Dokumentation als Kontext für KI-Modelle könnte die zentrale Herausforderung des DDD hinsichtlich der Kosten-Nutzen-Abwägung verbessern. Indem die detaillierte Vorarbeit in der Dokumentation direkt in Code umgesetzt wird, könnte der initiale Mehraufwand kompensiert und der Nutzen maximiert werden.

Dies könnte einen lang gehegten Traum der Softwareentwickler realisieren: die automatisierte Generierung von Quellcode [Mas+23]. Die Entwickler würden verstärkt als „Sys-

temdenker“ agieren, indem sie das Verhalten von Systemen spezifizieren, die KI-Tools steuern und überwachen und bei Bedarf eingreifen [Kal24].

3.2.1 Verwendung der Dokumentation als Kontext für KI-Systeme

Die Idee, Dokumentation als direkte Grundlage für die KI-basierte Codegenerierung zu nutzen, ist relativ neu und wurde nicht umfassend erforscht. Ein bemerkenswerter Beitrag stammt von Chemnitz u. a. [Che+23], die eine Methode vorschlagen, um Anwendungs-Code aus Testfallspezifikationen zu generieren und so den Zeitaufwand für Entwickler in der Anwendungsentwicklung weiter zu reduzieren. Ihr Ansatz zielt darauf ab, relevante Informationen aus Testfallspezifikationen zu extrahieren, die den Standards des Behavior-Driven Development (BDD) entsprechen. BDD ist eine Technik, die sich aus dem TDD entwickelt hat und ein agiler Softwareentwicklungsprozess ist. Im Kern konzentriert sich BDD darauf, Testfälle zu Beginn eines Entwicklungszyklus gemeinsam mit Domänenexperten und Stakeholdern zu spezifizieren [Che+23; SMT23]. Um sicherzustellen, dass diese Testfälle für alle Beteiligten verständlich sind, werden sie häufig in einer Domain-Specific Language (DSL) verfasst, die natürliche Sprache integriert und von spezialisierten Tools oder Frameworks wie Cucumber[6] ausgeführt wird.

Während dieser Ansatz bereits zeigt, wie Dokumentation zur Codegenerierung genutzt werden kann, geht das KI-gestützte DDD noch einen Schritt weiter. Die Kernidee besteht darin, die vorhandene Dokumentation sowie andere relevante Informationen als Kontextinformationen für KI-Tools zu verwenden, um funktionierenden Code zu generieren. Mit Abbildung 3.2 wird die vom Autor im Rahmen dieser Bachelorarbeit vorgeschlagene Weiterentwicklung des von Supalla [Sup14] beschriebenen Ansatzes visualisiert, der bereits in Abschnitt 2.2 eingeführt wurde:

[6] https://cucumber.io/

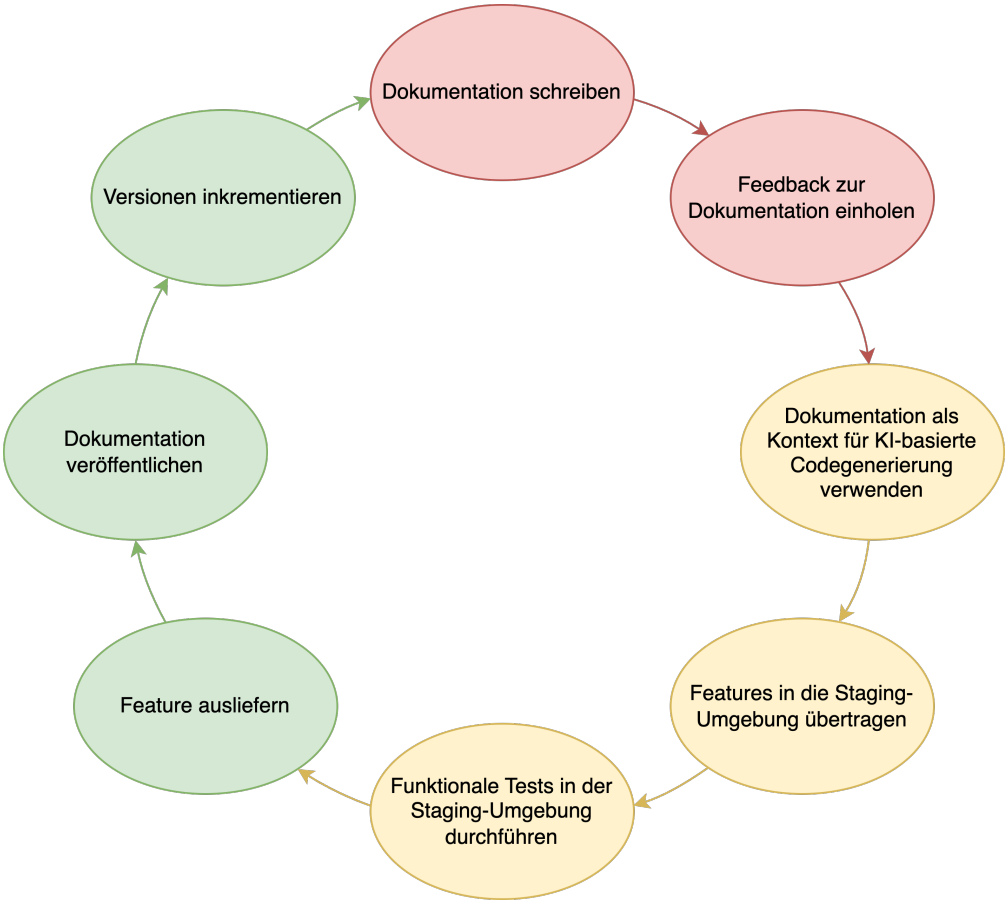

Abbildung 3.2: Vorgehensweise in KI-gestütztem DDD, eigene Darstellung

Der Unterschied liegt vor allem in der Phase „Nach Dokumentation testen und implementieren“ (auf der Abbildung mit gelb dargestellt). Anstelle des reinen TDD-Ansatzes wird die Dokumentation als Kontext für die KI-basierte Codegenerierung verwendet. Dieser Prozessschritt folgt unmittelbar nach der Dokumentationserstellung (in der Abbildung rot markiert). Hierbei kann das in Abschnitt 3.1.3 erläuterte Prompt Pattern „Kontextmanager“ eingesetzt werden, um dem LLM gezielt nur die relevanten Kontextinformationen bereitzustellen. Der TDD-Ansatz könnte dennoch komplementär eingesetzt werden, wobei die Tests den KI-Tools als zusätzliche Kontextinformationen für die Codegenerierung dienen. Abschließend wird der KI-generierte Code überprüft und durch weitere gezielte Prompts im Bedarfsfall optimiert, um eine vollständige Erfüllung der Anforderungen sicherzustellen.

Es wird keine spezifische Art, kein spezifisches Format oder Formalisierungsgrad für die Dokumentation vorgeschrieben. Im Grunde gilt alles, was dazu dient, Gedanken zu ei-

nem Thema zu kommunizieren, als Dokumentation [Cru22]. Es ist jedoch entscheidend, dass die Dokumentation möglichst einheitlich ist. Idealerweise sollte sie den Richtlinien und Standards folgen, die innerhalb des Unternehmens, in den Teams oder in anderen relevanten Organisationseinheiten festgelegt sind. Dies gewährleistet, dass alle Entwickler wissen, wie die Dokumentation zu erstellen ist, und trägt dazu bei, den Lernaufwand für das LLM zu minimieren, da es sich nicht ständig auf neue Darstellungsweisen einstellen muss.

Die Quellcode-Dokumentation spielt eine zentrale Rolle in diesem Ansatz, weil sie in der ersten Linie von KI-Tools analysiert wird. Laut Rai, Belwal und Gupta [RBG22] kann die Dokumentation auf verschiedenen Ebenen erstellt werden, abhängig von der Programmiersprache, dem Framework und den spezifischen Anforderungen (siehe Abbildung 3.3). Beispielsweise erlaubt Python zusätzlich, Dokumentationen für gesamte Module in `__init__.py`-Dateien zu schreiben.

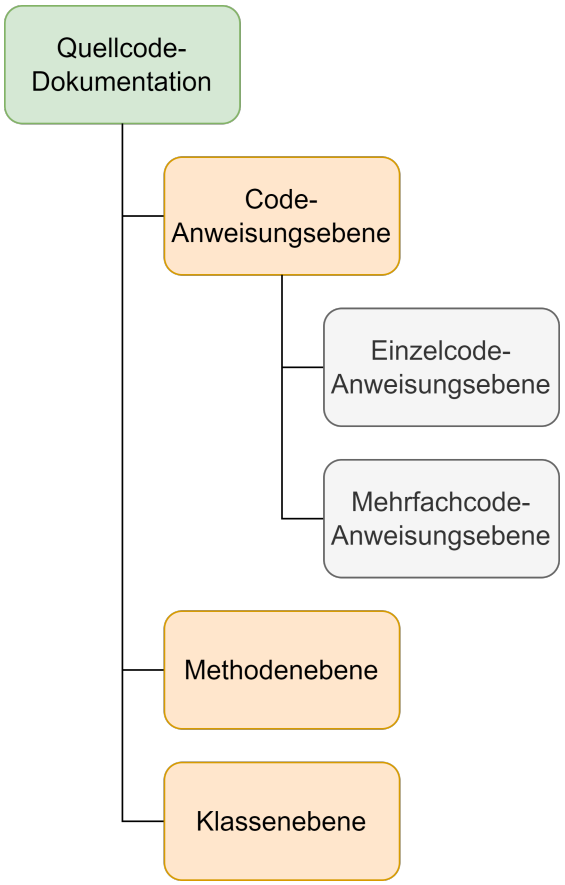

Abbildung 3.3: Ebene der Quellcode-Dokumentation nach [RBG22], eigene Darstellung

Es ist von entscheidender Bedeutung, dass die Dokumentation genau beschreibt, welche Funktionalität der dazugehörige oder zu generierende Quellcode-Block haben muss. Die Beschreibung sollte klar und präzise sein, um sowohl dem KI-Tool als auch anderen Entwicklern ein eindeutiges Verständnis der Anforderungen zu vermitteln. Die Dokumentation richtet sich in erster Linie an andere Softwareentwickler, die mit dem Code arbeiten werden.

Beispiel für die Integration von DDD und KI-gestützter Codegenerierung

Ein hervorragendes Beispiel ist die Funktion `fetch_bigtable_rows` von Google [Goo24], die in Abbildung 3.4 dargestellt ist.

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.

    Args:
        big_table: An open Bigtable Table instance.
        keys: A sequence of strings representing the key of each table row
            to fetch.
        other_silly_variable: Another optional variable, that has a much
            longer name than the other args, and which does nothing.

    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    Raises:
        IOError: An error occurred accessing the bigtable.Table object.
    """
```

Abbildung 3.4: Beispiel für Verwendung von Dokumentation als Kontext für KI-gestützte Codegenerierung nach [Goo24], eigene Darstellung

In diesem Beispiel ist die Docstring-Dokumentation der Funktion `fetch_bigtable_rows` ausführlich und strukturiert gemäß den Google Style Guidelines verfasst. Sie enthält:

- **Eine klare Beschreibung der Funktion**: Was die Methode tut und welchen Zweck sie erfüllt.

- **Detaillierte Angaben zu den Argumenten (`Args`)**: Erklärung jedes Parameters, einschließlich Datentyp und Bedeutung.
- **Informationen über den Rückgabewert (`Returns`)**: Beschreibung dessen, was die Funktion zurückgibt, mit Beispielen zur Veranschaulichung.
- **Angabe möglicher Ausnahmen (`Raises`)**: Auflistung von Fehlern, die während der Ausführung auftreten können.

Diese umfassende Dokumentation dient als effektives Prompt für LLM, um den entsprechenden Code zu generieren. Indem sowohl die Docstrings als auch die Funktionssignatur bereitgestellt werden, erhält das KI-Tool einen klaren Kontext und kann den Code innerhalb dieser Vorgaben erstellen [OBr+24].

Falls erforderlich, können zusätzliche Dokumentationen in den Prompt aufgenommen werden, um weiteren Kontext zu bieten. Dies könnten Hintergrundinformationen zum Projekt, spezifische Geschäftslogiken oder technische Spezifikationen sein, die für die Generierung des Codes relevant sind, aber nicht unbedingt in die Dokumentation des spezifischen Code-Blocks gehören.

Mögliche Herausforderungen sowie Lösungsansätze

Die Anwendung der Vorgehensweise kann idealerweise dazu führen, dass aus der vorhandenen Dokumentation funktionierender Code generiert wird, wodurch der Aufwand für das manuelle Schreiben des Codes erheblich reduziert – das heißt, der Entwickler muss lediglich wenige Teile des generierten Codes gezielt anpassen - oder sogar vollständig entfällt. Vorausgesetzt, die Qualität der Dokumentation ist hoch - was durch die Praktiken des DDD gefördert werden kann (siehe Abschnitt 2.3.4) - ergeben sich dennoch einige Herausforderungen bei der Umsetzung dieses Ansatzes.

Vollständigkeit des Kontextes Ein zentrales Problem besteht darin, dass Kontext für die Codegenerierung von entscheidender Bedeutung ist [Sta24]. Fehlen bestimmte Informationen oder ist die Dokumentation nicht ausreichend detailliert, kann das KI-Modell Code erzeugen, der nur mit erheblichen Anpassungen in das bestehende Projekt integriert werden kann. Dies kann dazu führen, dass Entwickler zusätzlichen Aufwand betreiben müssen, um den generierten Code anzupassen und zu korrigieren, was den erwarteten Effizienzgewinn mindert.

Dieses fehlende Projektwissen kann teilweise durch Fine-Tuning des KI-Modells behoben werden. Fine-Tuning beinhaltet das Training des LLMs auf einer großen Menge zusätzlicher Daten, um die generierten Antworten besser an die spezifischen Bedürfnisse anzupassen. Park u. a. [Par+23] erklären, dass Fine-Tuning insbesondere bei der Erstellung von konversationellen Modellen wie Chatbots eingesetzt wird und eine erhebliche Menge an Trainingsdaten erfordert. Für Unternehmen, die über eine umfangreiche Sammlung gut dokumentierten Codes verfügen, bietet sich hier die Möglichkeit, diese Ressourcen zu nutzen, um das KI-Modell gezielt auf ihre spezifischen Anforderungen zu konfigurieren[7].

In Fällen, in denen ein Unternehmen eine proprietäre Programmiersprache oder spezielle Frameworks verwendet, die dem LLM nicht bekannt sind, wird das Fine-Tuning noch wichtiger. Davila u. a. [Dav+24] betonen, dass das Anpassen des Modells an solche speziellen Technologien entscheidend ist, um effektive Ergebnisse zu erzielen. Ohne dieses spezifische Training kann das KI-Modell möglicherweise keinen brauchbaren Code generieren oder Fehler produzieren, die zusätzliche Korrekturen erfordern.

Ein weiterer Vorteil des Fine-Tunings liegt in der Verbesserung der Sicherheit des generierten Codes. Li u. a. [Li+24] zeigen, dass das Fine-Tuning mit einem Datensatz von behobenen Sicherheitslücken die Leistung des Modells bei der Generierung nicht verwundbarer Codes um 10% für die Programmiersprache C verbessern kann. Dies ist besonders relevant, da KI-Modelle, die auf öffentlich verfügbaren Codes trainiert wurden, potenziell auch bestehende Sicherheitslücken und Bugs reproduzieren können [Maj+24].

Qualität der Dokumentation Wenn die Qualität der Dokumentation oder des vorhandenen Codes nicht ausreichend oder gar unzureichend ist, sollte diese vorab verbessert werden. Bei großen und älteren Projekten kann dies jedoch eine erhebliche Herausforderung darstellen. Ohne eine gründliche Bereinigung und Aktualisierung der Dokumentation besteht das Risiko, dass das LLM von schlechten Beispielen lernt und fehlerhaften oder unbrauchbaren Code generiert. Dies kann den Aufwand für die Entwickler sogar erhöhen, da sie mehr Zeit für die Überprüfung und Korrektur des generierten Codes aufwenden müssen [Cou+24].

[7] OpenAI stellt beispielsweise eine dedizierte Fine-Tuning-Schnittstelle für ihre Modelle bereit: https://platform.openai.com/docs/guides/fine-tuning

Qualität des generierten Codes Ein weiterer wesentlicher Aspekt ist die Qualität des generierten Codes. Rodriguez [Rod23] definiert Codequalität anhand eines Kriterienkatalogs, der wissenschaftliche und industrielle Standards vereint:

- **Lesbarkeit**: Der Code folgt sprachspezifischen Idiomen und Namenskonventionen. Unverständlicher Code erschwert Wartung, Verbesserung und Dokumentation.
- **Wiederverwendbarkeit**: Durch modulare Gestaltung lässt sich Code effizient in verschiedenen Kontexten einsetzen, was Zusammenarbeit erleichtert, Ressourcen spart und für Konsistenz sorgt.
- **Kürze**: Das Don't Repeat Yourself (DRY)-Prinzip hilft, Redundanzen zu vermeiden, die Lesbarkeit zu verbessern und potenzielle Fehlerquellen zu reduzieren.
- **Wartbarkeit**: Eine klare Struktur, geringe Abhängigkeiten und eine transparente Funktionalität fördern eine einfache Weiterentwicklung und erleichtern die Wiederverwendung von Code.
- **Resilienz**: Fehlerantizipation und -behandlung gewährleisten stabile Funktionalität auch bei Störungen.

Da der Code jedoch generiert wird, sollte ein weiteres Kriterium berücksichtigt werden:

- **Funktionale Korrektheit**: Der erzeugte Code entspricht den übergebenen Anforderungen.

Selbst wenn das KI-Tool über ausreichend Kontext verfügt und sich die Leistungsfähigkeit von LLMs mit jeder neuen Version verbessert [Cla+24], können im erzeugten Code dennoch Bugs oder andere Fehler auftreten. Clark u. a. [Cla+24] stellen fest, dass im Durchschnitt zwar qualitativ hochwertigen Code generiert wurde und dabei eine gewisse Konsistenz aufwies, gab es jedoch gelegentlich Fehler ähnlich denen von menschlichen Entwicklern. Laut Stack Overflow [Sta24] sind Entwickler in Bezug auf die Genauigkeit von KI-Ausgaben geteilter Meinung: 43% der Befragten haben Vertrauen in die Genauigkeit von KI, während 31% skeptisch sind. Besonders bei komplexen Aufgaben glauben fast die Hälfte (45%) der professionellen Entwickler, dass KI-Tools schlecht oder sehr schlecht darin sind, solche Herausforderungen zu bewältigen. Dies unterstreicht die Notwendigkeit, KI-generierten Code kritisch zu prüfen und nicht uneingeschränkt darauf zu vertrauen.

Nicht-Determinismus des generierten Codes Ein weiteres Merkmal von KI-Tools wie GitHub Copilot ist ihr Nicht-Determinismus. Das bedeutet, dass bei jeder Codegenerierung für dasselbe Prompt unterschiedliche Ergebnisse entstehen können [Maj+24]. Auch Clark u. a. [Cla+24] beobachteten, dass die Konsistenz der generierten Codequalität zwar insgesamt stabil ist, aber in bestimmten Metriken variieren kann.

Aufgrund dieses Nicht-Determinismus können Entwickler versuchen, das LLM den Code mit dem gleichen Prompt mehrfach generieren zu lassen und anschließend die beste Variante auszuwählen. Dieses Vorgehen erhöht die Wahrscheinlichkeit, einen Code zu erhalten, der den Anforderungen entspricht und weniger Fehler aufweist. Wenn der generierte Code trotz mehrfacher Versuche nicht den Erwartungen entspricht, besteht die Möglichkeit, das Prompt im Dialog mit dem KI-Tool zu verfeinern. Durch gezielte Anpassungen und Klarstellungen im Prompt kann das Modell besser auf die spezifischen Anforderungen ausgerichtet werden.

Zusätzlich kann eine manuelle Anpassung des Codes notwendig sein. Trotz der Unterstützung durch KI-Tools bleibt die Expertise von Entwicklern unverzichtbar, um sicherzustellen, dass der Code funktional korrekt ist und den Qualitätsstandards entspricht Majdinasab u. a. [Maj+24].

4 Proof of Concept des KI-gestützten DDD

Um die in den Kapiteln 2 und 3 beschriebene Vorgehensweise praktisch zu demonstrieren, wird in diesem Kapitel eine Beispielanwendung implementiert. Ziel ist es, die theoretischen Konzepte in einem realitätsnahen Kontext anzuwenden und die Prinzipien des KI-gestützten DDD zu veranschaulichen. Der Fokus liegt hierbei weniger auf der Erstellung einer produktionsreifen Anwendung, sondern vielmehr auf der Demonstration des Softwareentwicklungsprozesses als solchem.

Die Untersuchung erfolgt anhand von zwei ausgewählten Szenarien, die typischerweise in der Softwareentwicklung vorkommen:

1. Initialisierung des Projekts.
2. Erweiterung der Anwendung durch Hinzufügen neuer Anforderungen.

In jedem dieser Szenarien wird demonstriert, wie die Entwicklung unter Anwendung des KI-gestützten DDD abläuft. Insbesondere wird gezeigt, wie die Dokumentation organisiert wird und wie sie als Grundlage für die KI-gestützte Codegenerierung dient. Dabei wird das in Abschnitt 3.2.1 beschriebene Verwendung der Dokumentation als Kontext umgesetzt.

4.1 Szenario 1: Initialisierung des Projekts

Dieses Szenario zeigt, wie ein Softwareprojekt mit Hilfe des KI-gestützten DDD gestartet wird. Es umfasst die Entwicklung der Idee, die Auswahl der Tools und Technologien, die Organisation der Dokumentation sowie die Definition der Vorgehensweise, die während des gesamten Projekts befolgt wird.

4.1.1 Idee des Proof of Concept

Die Einführung in die Anwendung erfolgt in der Datei `NumSort.md` – hierzu liefert Abbildung 4.1 eine anschauliche Darstellung.

```
1  # NumSort
2
3  NumSort is a command-line application designed to sort
   lists of numbers using a selection of well-known in-place
   sorting algorithms.
4  The application accepts input in the form of CSV
   (Comma-Separated Values) files and outputs the sorted data
   in the same format.
5  NumSort serves as an educational tool, allowing users to
   explore and understand how different sorting algorithms
   work in practice.
6
7  ## Table of Contents <!-- omit in toc -->
8
   Table of Contents (up to date)
9  - [Idea](#idea)
10 - [Goals](#goals)
11 - [Stakeholders](#stakeholders)
12   - [Students](#students)
13   - [Teachers](#teachers)
```

Abbildung 4.1: Einführung und Inhaltsverzeichnis der `NumSort.md`, eigene Darstellung

Die Idee für die Anwendung namens „NumSort“ besteht darin, eine kleine Konsolenanwendung zu entwickeln, die eine Liste von Zahlen mit Hilfe ausgewählter bestehender In-Place-Sortieralgorithmen sortiert. Als Ein- und Ausgabeformate werden Comma-Separated Values (CSV)-Dateien verwendet, was eine einfache Integration mit anderen Anwendungen und Datenquellen ermöglicht. Die zugehörige Dokumentation wird in Abbildung 4.2 veranschaulicht.

```
15  ## Idea
16
17  The core idea behind NumSort is to provide a simple yet
    powerful platform for learning and experimenting with
    sorting algorithms.
18  By implementing the algorithms in a command-line interface
    (CLI) application, users can interact with the sorting
    processes directly from the console.
19  The application is intended to be both a learning aid for
    students and a teaching resource for educators.
20
21  NumSort is built using Python and leverages libraries such
    as Click for creating user-friendly CLI interfaces and
    tqdm for displaying progress bars during sorting
    operations.
22  The application is documented thoroughly using Markdown,
    and architectural diagrams are provided using draw.io to
    facilitate understanding of the codebase and system design.
```

Abbildung 4.2: Dokumentation der Idee von NumSort, eigene Darstellung

Die Hauptziele der Anwendung sind:

- **Lernzwecke**: NumSort soll als pädagogisches Werkzeug dienen, um das Verständnis von Sortieralgorithmen zu fördern. Benutzer können verschiedene Algorithmen anwenden und ihre Funktionsweise anhand von Eingabedaten nachvollziehen.
- **Erweiterbarkeit**: Es besteht die Möglichkeit, die Anwendung zukünftig zu erweitern, um die Arbeitsweise der Algorithmen zu veranschaulichen, beispielsweise durch Visualisierung der Sortierschritte oder Leistungsanalysen.

Diese Ziele werden in Abbildung 4.3 übersichtlich dargestellt.

```
24 ## Goals
25
26 The primary goals of NumSort are:
27
28 1. **Educational Value**: To help users learn how
   different in-place sorting algorithms function by allowing
   them to see the algorithms in action with their own data
   sets.
29 2. **Ease of Use**: To provide a straightforward
   command-line interface that can be easily used without
   prior extensive knowledge of programming or CLI tools.
30 3. **Extensibility**: To offer a codebase that is
   well-documented and structured, making it easy for other
   developers to extend the application with additional
   sorting algorithms or features.
31 4. **Comparison and Analysis**: To enable users to compare
   the performance and behavior of various sorting algorithms
   under different conditions and data sets.
32 5. **Integration with AI Tools**: To explore how
   AI-assisted coding tools, such as GitHub Copilot, can aid
   in the development process, particularly in implementing
   standard algorithms and creating custom CLI applications.
33
34 By providing a practical implementation of sorting
   algorithms in a user-friendly format, NumSort aims to
   bridge the gap between theoretical algorithm study and
   practical application.
35 The application not only facilitates learning but also
   encourages exploration and experimentation, making it a
   valuable resource for both students and educators in the
   field of computer science.
```

Abbildung 4.3: Dokumentation der Ziele von NumSort, eigene Darstellung

Im Anschluss wird erörtert, wer die Hauptstakeholder der Anwendung sind. Für NumSort stehen Studierende und Lehrende im Mittelpunkt, die sich mit Sortieralgorithmen auseinandersetzen – dies wird in Abbildung 4.4 ebenfalls veranschaulicht.

```
37 ## Stakeholders
38
39 ### Students
40
41 - **Learning Algorithms**: Students studying computer
   science or related fields can use NumSort to deepen their
   understanding of sorting algorithms by observing how each
   algorithm processes data step by step.
42 - **Hands-On Experience**: Provides an opportunity for
   practical engagement, allowing students to modify and
   experiment with the code to see the effects of their
   changes.
43 - **Project Foundation**: Serves as a starting point for
   academic projects or assignments where students can add
   new features or algorithms as part of their coursework.
44
45 ### Teachers
46
47 - **Teaching Aid**: Educators can use NumSort as a
   teaching tool in lectures or labs to demonstrate the
   implementation and performance of different sorting
   algorithms.
48 - **Curriculum Integration**: Can be integrated into
   course materials, providing a practical example that
   complements theoretical lessons on algorithms.
49 - **Customization**: Instructors can modify the
   application to suit specific teaching objectives, such as
   highlighting particular algorithmic concepts or
   introducing complexity analysis.
```

Abbildung 4.4: Dokumentation der Stakeholder von NumSort, eigene Darstellung

Die Implementierung der Sortieralgorithmen basiert auf den Lernmaterialien von freeCodeCamp [fre19]. Diese Materialien wurden ausgewählt, weil sie

- hohe Qualität der Inhalte bieten: Die Beschreibungen und Anleitungen von freeCodeCamp[8] sind umfassend und wurden von einer breiten Entwicklergemeinschaft überprüft.
- praxisnahe Beispielimplementierungen enthalten: Die bereitgestellten Codebeispiele ermöglichen einen direkten Vergleich mit dem von GitHub Copilot generierten Code, was eine Bewertung der Qualität und Effizienz des KI-generierten Codes erlaubt.

[8] https://www.freecodecamp.org/news/about/

4.1.2 Tools und Technologien

Für die Umsetzung der Beispielanwendung werden die folgenden Tools und Technologien eingesetzt:

- **Python 3.12**: Eine Programmiersprache, die für die Implementierung der Anwendung verwendet wird. Python ist aufgrund seiner Einfachheit und der umfangreichen Standardbibliothek eine weit verbreitete Wahl für Prototyping und Entwicklung.
- **Click 8.1.7**[9]: Eine Python-Bibliothek zur Erstellung von Command-Line Interfaces (CLIs). Sie erleichtert die Entwicklung von benutzerfreundlichen und gut strukturierten Kommandozeilenanwendungen.
- **tqdm 4.66.5**[10]: Eine Python-Bibliothek zur Anzeige von Fortschrittsbalken in der Konsole. Sie ermöglicht es, den Fortschritt von iterativen Prozessen übersichtlich darzustellen.
- **Markdown**: Ein leichtgewichtiges Markup-Format zur Erstellung von gut lesbarer und strukturierter Dokumentation. Es wird zur Dokumentation der Anwendung und ihrer Funktionen verwendet.
- **draw.io**[11]: Ein Online-Tool zur Erstellung von Diagrammen und Flussdiagrammen. Es dient zur Visualisierung der Architektur und des Designs der Anwendung.
- **Visual Studio Code (VS Code)**[12]: Ein vielseitiger und erweiterbarer Code-Editor, der für die Entwicklung genutzt wird. Er bietet umfangreiche Funktionen wie Syntax-Highlighting, Debugging und Unterstützung für Erweiterungen.
- **Git**: Ein Versionsverwaltungssystem zur Nachverfolgung von Änderungen am Quellcode. Es wird verwendet, um die Nachvollziehbarkeit bei der Entwicklung zu gewährleisten.
- **GitHub**[13]: Eine Plattform zum Hosting von Git-Repositories. Sie bietet CI/CD-Integration, die für die Automatisierung von Tests und Bereitstellung genutzt wird.

[9] https://palletsprojects.com/projects/click/
[10] https://tqdm.github.io/
[11] https://www.drawio.com/
[12] https://code.visualstudio.com/
[13] https://github.com

- **GitHub Copilot**: Ein KI-gestütztes Werkzeug zur Codegenerierung. Es nutzt maschinelles Lernen, um auf Basis von Kommentaren und Codefragmenten Vorschläge für Code zu machen. Es ist de facto Stand der Technik im Bereich der Codegenerierung [Mas+23; Sta24].

4.1.3 Organisation der Dokumentation

Nachdem die ersten Gedanken zur Anwendung gesammelt wurden, ist es essenziell, diese ausführlich zu dokumentieren und zu planen, wie die Dokumentation organisiert wird. Eine klare und gut gepflegte Dokumentation erleichtert nicht nur die Entwicklung, sondern auch die spätere Wartung und Erweiterung der Anwendung [DAD06; GWW23].

Basierend auf der in Kapitel 2 beschriebenen Kategorisierung wurde eine Ordnerstruktur entworfen, die die verschiedenen Dokumentationstypen systematisch erfasst. Die gewählte Struktur für NumSort ist in Abbildung 4.5 dargestellt.

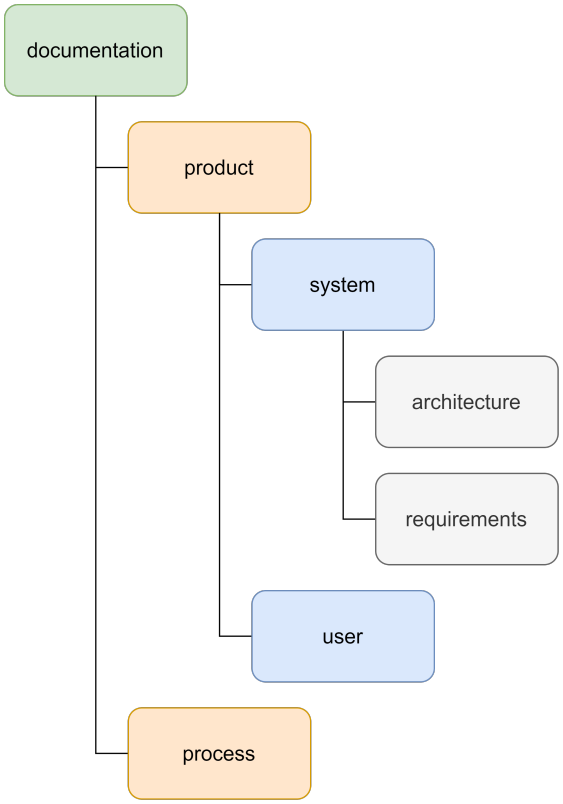

Abbildung 4.5: Organisation der Software-Dokumentation in NumSort, eigene Darstellung

Diese Struktur ermöglicht eine klare Trennung zwischen Prozess- und Produktdokumentation sowie eine weitere Unterteilung der Produktdokumentation in Benutzerdokumentation und Systemdokumentation. Innerhalb der Systemdokumentation dienen die Unterordner `architecture` und `requirements` dazu, Architekturbeschreibungen und Anforderungsdokumente strukturiert abzulegen.

Alle diese und weitere Bestandteile des Repositories sind in der `README.md`-Datei beschrieben. Ihre Inhalte sind in Abbildung 4.6 dargestellt. Diese Datei fungiert als zentraler Einstiegspunkt und bietet eine Übersicht über das gesamte Repository. Sie erleichtert neuen Entwicklern und Benutzern den Einstieg und dient als Referenz für die Projektstruktur [Pre10].

```
1  # NumSort
2
3  NumSort is a command-line application that sorts lists of
   numbers using a selection of well-known in-place sorting
   algorithms.
4  It accepts input in CSV format and outputs the sorted data
   similarly, facilitating easy integration with other tools.
5  Designed as an educational tool, NumSort helps students
   and educators explore and understand how different sorting
   algorithms function in practice.
6  Built with Python, it is easy to use and extend, making it
   suitable for learning, teaching, and further development
   by developers.
7
8  ## Table of Contents
9
   Table of Contents (up to date)
10 - [Repository Overview](#repository-overview)
11 - [Prerequisites](#prerequisites)
```

Abbildung 4.6: Einführung und Inhaltsverzeichnis der `README.md`, eigene Darstellung

Software-Dokumentation kann in einer Vielzahl von Formaten vorliegen, die je nach Kontext und Bedarf der Entwickler oder Stakeholder unterschiedlich genutzt werden. Diese Formate reichen von traditionellen Textdokumenten wie Word- oder Portable Document Format (PDF)-Dateien bis hin zu moderneren, interaktiven Webplattformen und wikis, die speziell dafür entwickelt wurden, Architektur und andere technische Details zu dokumentieren [Mus+22; Ros+13; GWW23]. Aufgrund der überschaubaren Größe der Anwendung wurde Markdown als primäres Format für die Dokumentation gewählt. Markdown

bietet eine einfache und intuitive Syntax, die sowohl von Entwicklern als auch von Nicht-Entwicklern leicht erlernt und genutzt werden kann. Zudem ermöglicht es eine schnelle Bearbeitung und Versionierung der Dokumentation direkt im Git-Repository.

Sollte das Projekt an Umfang zunehmen, kann das Dokumentationsformat entsprechend angepasst werden. Beispielsweise könnten aus den Markdown-Dateien mittels Tools wie MkDocs[14] oder Sphinx[15] automatisiert Webseiten generiert werden, um eine benutzerfreundlichere Darstellung zu ermöglichen.

Eine Ausnahme bildet die Quellcode-Dokumentation. Hier werden Docstrings innerhalb der Python-Dateien verwendet, um Klassen und Funktionen/Methoden direkt im Code zu dokumentieren. Dabei wird der Dokumentationsstil von Google [Goo24] angewendet, der klare Richtlinien für die Struktur und Formatierung von Docstrings bietet und eine einheitliche und lesbare Codebasis fördert.

4.1.4 Definition der Vorgehensweise

Für Entwickler ist es von großem Vorteil, wenn die angewandte Vorgehensweise klar dokumentiert ist. Eine gut strukturierte Prozessbeschreibung ermöglicht es, einer eindeutigen Abfolge von Schritten zu folgen, um eine bestimmte Funktionalität zu implementieren. Dies kann viele Fragen im Voraus klären, Missverständnisse vermeiden und den Entwicklungsprozess effizienter gestalten.

Für NumSort wird die folgende Entwicklungsschrittfolge definiert:

1. Anforderung erstellen.
2. Benutzerdokumentation aktualisieren.
3. Architekturdokumentation aktualisieren.
4. Quellcode-Dokumentation erstellen.
5. Tests erstellen.
6. Anforderung implementieren.
7. Bereitstellung.

[14] https://www.mkdocs.org/
[15] https://www.sphinx-doc.org/en/master/

Sie ist auf Basis der in Abschnitt 2.2 beschriebenen Vorgehensweise strukturiert. Detailliertere Informationen zu jedem dieser Schritte sind in der Datei `Development Process.md` festgehalten (siehe Abbildung 4.7).

```
1  # Development Process
2
3  The development process for NumSort is meticulously
   structured to ensure clarity, consistency, and efficiency.
4  By following a well-defined set of steps, developers can
   seamlessly implement new features, fix bugs, and enhance
   the application while maintaining high-quality code and
   comprehensive documentation.
5  This procedure serves as a guideline for both current and
   future contributors, facilitating collaboration and
   simplifying the onboarding process for new developers.
6
7  ## Table of Contents <!-- omit in toc -->
8
   Table of Contents (up to date)
9  - [Implementation Steps](#implementation-steps)
10   - [1. Create Requirement](#1-create-requirement)
11   - [2. Update User Documentation]
     (#2-update-user-documentation)
12   - [3. Update Architecture Documentation]
     (#3-update-architecture-documentation)
13   - [4. Create Source Code Documentation]
     (#4-create-source-code-documentation)
14   - [5. Create Tests](#5-create-tests)
15   - [6. Implement Requirement](#6-implement-requirement)
16   - [7. Deployment](#7-deployment)
```

Abbildung 4.7: Einführung und Inhaltsverzeichnis der `Development Process.md`, eigene Darstellung

Abbildung 4.8 veranschaulicht beispielhaft die Dokumentation des Entwicklungsschrittes „Anforderung erstellen“.

```
18  ## Implementation Steps
19
20  The following steps outline the process of implementing a
    new feature in NumSort, from requirement definition to
    deployment.
21  For changing an existing feature or fixing a bug, the same
    steps can be adapted accordingly.
22
23  ### 1. Create Requirement
24
25  The first step involves clearly defining the requirement
    that needs to be addressed.
26  Each requirement should consist of:
27
28  - **Description of the Requirement:**
29    - A detailed explanation of the feature or functionality
    to be implemented.
30  - **Acceptance Criteria:**
31    - Specific criteria that must be met for the requirement
    to be considered complete.
32    - Any constraints or dependencies that may affect
    implementation.
33  - **Example Use Cases:**
34    - Practical scenarios illustrating how the feature will
    be used.
35    - Examples that highlight the user interaction with the
    application.
36    - Edge cases or exceptional conditions to consider.
37
38  *Example:*
39
40  See [REQ-1.md](../product/system/requirements/REQ-1.md)
    for an example requirement definition.
```

Abbildung 4.8: Dokumentation des Entwicklungsschrittes „Anforderung erstellen“, eigene Darstellung

4.1.5 Ergebnisse des ersten Szenarios

Das erste Szenario führt zur erfolgreichen Initialisierung des Projekts. Ziel ist es, dass alle Beteiligten ein gemeinsames Verständnis darüber entwickeln, welche Ziele das Projekt verfolgt, wie es strukturiert ist und wie der Entwicklungsprozess ablaufen soll, wobei dies

in der Dokumentation festgehalten wird. Den aktuellen Zwischenstand von NumSort nach der Initialisierungsphase zeigt Abbildung 4.9.

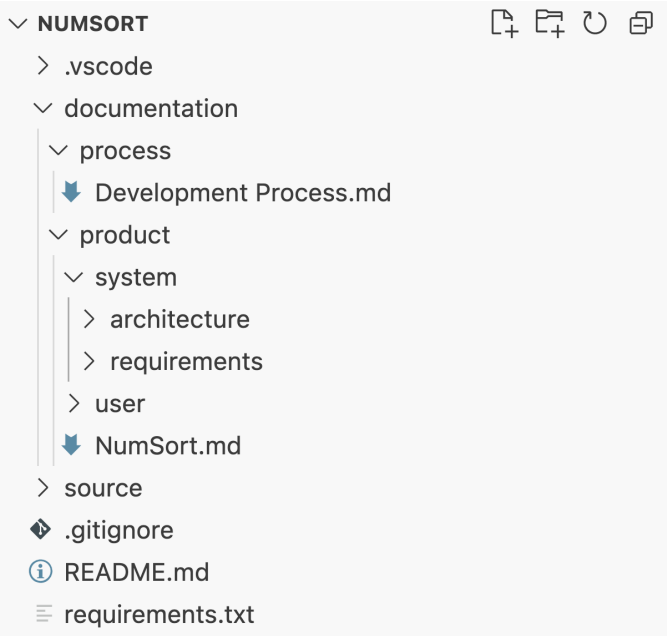

Abbildung 4.9: Stand von NumSort nach Implementierung des Szenarios 1, eigene Darstellung

Bisher wurde noch kein Quellcode geschrieben - dies erfolgt erst im nächsten Szenario.

4.2 Szenario 2: Erweiterung der Anwendung durch Hinzufügen neuer Anforderungen

In diesem Szenario werden die in Abschnitt 4.1.4 definierten Schritte für jede der folgenden Anforderungen angewendet, wobei die Erweiterung der Anwendung durch das Hinzufügen einer neuen Anforderung in diesem Beispiel eine Iteration im Entwicklungsprozess darstellt:

1. **`REQ-1` „Zahlen aus einer CSV-Datei einlesen“**: Die erste Anforderung bestand darin, eine CSV-Datei einzulesen und die enthaltenen Zahlen zu extrahieren. Diese Anforderung bildet die Grundlage für alle weiteren Funktionen der Anwendung, wie das Sortieren der eingelesenen Zahlen.

2. **`REQ-2` „Zahlen mit dem Bubble Sort-Algorithmus Sortieren“**: Der Sinn der zweiten Anforderung war es, den Bubble-Sort-Algorithmus in die Anwendung zu integrieren. Dieser Algorithmus wurde aufgrund seiner Einfachheit und Verständlichkeit gewählt, was ihn ideal für Experimentierzwecke macht.

3. **`REQ-3` „Sortierte Zahlen in eine CSV-Datei schreiben“**: Die dritte Anforderung fügte der Anwendung die Fähigkeit hinzu, die sortierten Ergebnisse in eine CSV-Datei zu exportieren. Dies erhöht den Nutzen der Anwendung für die Endbenutzer, da die Ergebnisse nun für weitere Analysen oder Anwendungen genutzt werden können.

Dabei wird verdeutlicht, wie die Prinzipien des KI-gestützten DDD den Entwicklungsprozess unterstützen können. Die Ergebnisse jeder Iteration werden analysiert, um den Entwicklungsprozess zu optimieren und durch Anpassungen eine Verfeinerung und Verbesserung der weiteren Iterationen zu erzielen.

Da das Vorgehen bei allen drei Anforderungen identisch ist, wird im Folgenden nur die erste Anforderung `REQ-1` als Beispiel verwendet.

4.2.1 Anforderung erstellen

Im Entwicklungsprozess von NumSort ist es entscheidend, dass jede Anforderung klar und strukturiert definiert wird. Die detaillierte Beschreibung der `REQ-1`, einschließlich spezifischer Akzeptanzkriterien und Anwendungsbeispiele, ist in der Datei `REQ-1.md` zu finden. Dort werden alle relevanten Details erläutert, um eine reibungslose Implementierung und Validierung zu gewährleisten. Die Struktur der Anforderung wird in Abbildung 4.10 anschaulich dargestellt.

```
1  # REQ-1: Read Numbers from a CSV File
2
3  Implement the functionality for NumSort to read numbers
   from a CSV (Comma-Separated Values) file specified by the
   user.
4  The application should accept the CSV file as a
   command-line argument and parse its contents to retrieve
   the list of float numbers for sorting.
5
6  - The CSV file will contain only numbers separated by
   commas (e.g., `1,2,3.3,4,5`).
7  - The application should handle cases where the file does
   not exist or is empty by displaying appropriate error
   messages.
8  - This functionality lays the groundwork for subsequent
   features, such as sorting the numbers using various
   algorithms.
9
10 ## Table of Contents <!-- omit in toc -->
11
   Table of Contents (up to date)
12 - [Acceptance Criteria](#acceptance-criteria)
13 - [Example Use Cases](#example-use-cases)
14   - [Successful File Reading](#successful-file-reading)
15   - [Invalid Data Error](#invalid-data-error)
16   - [Empty File Error](#empty-file-error)
17   - [Unsupported Input Format](#unsupported-input-format)
```

Abbildung 4.10: Einführung und Inhaltsverzeichnis der `REQ-1.md`, eigene Darstellung

Um Einheitlichkeit und Klarheit zu gewährleisten, enthält jede Anforderung drei zentrale Komponenten:

- **Beschreibung der Anforderung**: Eine ausführliche Erklärung der Funktion oder des Features, das implementiert werden soll. Hierbei werden die Ziele, der Zweck und der Kontext der Anforderung dargelegt. Ein Beispiel für `REQ-1` ist in Abbildung 4.10 zu sehen.
- **Akzeptanzkriterien**: Klare und spezifische Kriterien, die erfüllt sein müssen, damit die Anforderung als erfolgreich umgesetzt gilt. Diese umfassen auch eventuelle Einschränkungen, Bedingungen oder Abhängigkeiten von anderen Systemkomponenten. Die ersten drei Akzeptanzkriterien der `REQ-1` werden in Abbildung 4.11 dargestellt.

```
19  ## Acceptance Criteria
20
21  - **Command-Line Argument:**
22    - The application accepts a single command-line argument
    specifying the input CSV file.
23    - Usage: `python numsort.py data.csv`
24  - **File Reading:**
25    - The application successfully opens and reads the
    contents of the specified CSV file.
26    - The file contains numbers separated by commas without
    any headers or additional text.
27  - **Data Parsing:**
28    - The application parses the CSV content and converts
    each value into a float number.
29    - Whitespace around the numbers should be appropriately
    handled (e.g., `1, 2, 3` is valid).
```

Abbildung 4.11: Erste drei Akzeptanzkriterien der REQ-1, eigene Darstellung

- **Beispiel-Use-Cases**: Praktische Anwendungsfälle, die veranschaulichen, wie die Funktion genutzt wird. Sie zeigen die Interaktion des Benutzers mit der Anwendung und berücksichtigen auch Sonderfälle oder Ausnahmen. Die ersten zwei Use-Cases der REQ-1 werden in Abbildung 4.12 veranschaulicht.

````
## Example Use Cases

### Successful File Reading

- **Description:**
  - A user has a CSV file named `data.csv` containing the numbers `5,3,1,4,2`.
  - They want to sort these numbers using NumSort.
- **User Interaction:**

  ```bash
  python numsort.py data.csv
  ```

- **Expected Outcome:**
  - The application reads the numbers successfully.
  - Proceeds to sorting (sorting functionality to be implemented in future requirements).

### Invalid Data Error

- **Description:**
  - The user's file `invalid.csv` contains `1,2,three,4,5`.
- **User Interaction:**

  ```bash
  python numsort.py invalid.csv
  ```

- **Expected Outcome:**
  - The application displays:

    ```bash
    Error: The file 'invalid.csv' contains invalid data.
    Expected numbers separated by commas.
    ```
````

Abbildung 4.12: Erste zwei Use-Cases der REQ-1, eigene Darstellung

Diese Struktur soll sicherstellen, dass die zu implementierende Funktionalität vollständig verstanden wird und dient als Grundlage für die Entwicklung und das Testen.

4.2.2 Benutzerdokumentation aktualisieren

Die Benutzerdokumentation für NumSort wurde entsprechend der ersten Anforderung strukturiert und aktualisiert, um Endbenutzern eine klare und verständliche Anleitung zu bieten. Die Dokumentation folgt der folgenden Struktur:

- **Kurze Einführung**: Eine Übersicht über die Anwendung, ihre Hauptfunktionen und den Nutzen für die Benutzer, wie in Abbildung 4.13 dargestellt.

```
1  # User Documentation (version 0.1.0)
2
3  > Please make sure you use the same release version as the
   documentation.
4
5  NumSort is a command-line application designed to sort
   lists of numbers using a variety of well-known in-place
   sorting algorithms.
6  It accepts input in the form of CSV (Comma-Separated
   Values) files and outputs the sorted data in the same
   format.
7  NumSort is ideal for users who need a simple and efficient
   tool to organize numerical data or for educational
   purposes to understand how different sorting algorithms
   work in practice.
8
9  With NumSort, you can:
10
11 - **Read numerical data from CSV files**: Easily input
   your data for sorting.
12 - **Choose from multiple sorting algorithms**: Select the
   algorithm that best suits your needs, such as Bubble Sort,
   Selection Sort, or Insertion Sort.
13 - **Specify output options**: Save the sorted data to a
   file or display it directly in the console.
14 - **Handle large datasets efficiently**: Sort extensive
   lists of numbers without compromising performance.
15 - **Experience user-friendly error messages**: Receive
   clear feedback if something goes wrong.
```

Abbildung 4.13: Einführung der Benutzerdokumentation, eigene Darstellung

- **Voraussetzungen**: Eine Liste der erforderlichen Software und Bibliotheken, um die Anwendung nutzen zu können. Die Benutzerdokumentation verweist auf `READ-`

`ME.md`, die alle Abhängigkeiten auflistet, um duplikative Informationen zu vermeiden. Die erste Voraussetzung wird in Abbildung 4.14 dargestellt.

````
31 ## Prerequisites
32
33 Before installing or running NumSort, please ensure that
   the following prerequisites are met:
34
35 - **Python 3.12 or higher**:
36   NumSort is developed using Python 3.12.
37   You need to have Python installed on your system to run
     the application.
38   If you do not have Python installed, you can download it
     from the [official Python website](https://www.python.
     org/downloads/).
39
40   To verify that Python is installed and check its
     version, run the following command in your terminal or
     command prompt:
41
42   ```bash
43   python --version
44   ```
45
46   The output should indicate Python 3.12 or a higher
     version.
````

Abbildung 4.14: Erste Voraussetzung der Benutzerdokumentation, eigene Darstellung

- **Installation**: Schritt-für-Schritt-Anweisungen zur Installation von NumSort.
- **Schnellstart**: Eine Anleitung, wie die gängigsten Funktionen der Anwendung verwendet werden.
- **Verfügbare Optionen**: Eine detaillierte Beschreibung aller verfügbaren Kommandozeilenoptionen.
- **Fehlerbehandlung**: Hinweise zu häufigen Fehlern und deren Behebung. Die ersten zwei Informationen werden in Abbildung 4.15 dargestellt.

````
91  ### Common Errors
92
93  #### File Not Found
94
95  - **Description**: The specified input file does not exist.
96  - **Error Message**:
97
98    ```text
99    Error: Invalid value for 'FILE_PATH': File 'data.csv'
      does not exist.
100   ```
101
102 - **Solution**: Check that the file name and path are
    correct.
103
104 #### Empty File
105
106 - **Description**: The input file is empty.
107 - **Error Message**:
108
109   ```bash
110   Error: File 'data.csv' is empty.
111   ```
112
113 - **Solution**: Ensure that the file contains numbers
    separated by commas.
````

Abbildung 4.15: Erste zwei Abschnitte der Benutzerdokumentation zur Fehlerbehandlung, eigene Darstellung

- **Unterstützte Sortieralgorithmen**: Eine Liste der in der Anwendung verfügbaren Sortieralgorithmen.
- **Hilfe und Support**: Informationen zur Kontaktaufnahme mit dem Entwicklerteam und zur weiteren Unterstützung.

Weitere detaillierte Informationen sind in der Datei `NumSort User Documentation.md` ausführlich beschrieben. Diese Datei dient als zentrale Ressource für Benutzer, um alle Funktionen und Aspekte der Anwendung zu verstehen.

Durch die Aktualisierung der Benutzerdokumentation wird klar definiert, wie die Anwendung nach der Implementierung der Anforderung funktionieren soll. Dies stellt sicher, dass die Entwicklung zielgerichtet erfolgt und die Erwartungen der Benutzer erfüllt werden. Zudem erleichtert eine gut strukturierte Dokumentation den Einstieg und die

effektive Nutzung der Anwendung für Endbenutzer ohne technisches Hintergrundwissen zur Implementierung.

4.2.3 Architekturdokumentation aktualisieren

Die Architekturdokumentation von NumSort wurde entsprechend der ersten Anforderung erstellt und strukturiert, um Entwicklern eine klare Übersicht über die Systemarchitektur zu bieten. Sie folgt der folgenden Gliederung:

- **Kurze Einführung**: Eine Übersicht, die erläutert, dass NumSort eine Konsolenanwendung ist, die in Python geschrieben wurde.
- **Building Block View**: Dieser Abschnitt enthält ein Komponentendiagramm, das die Hauptkomponenten der Anwendung visualisiert. Das Diagramm der `REQ-1` ist in Abbildung 4.16 dargestellt.

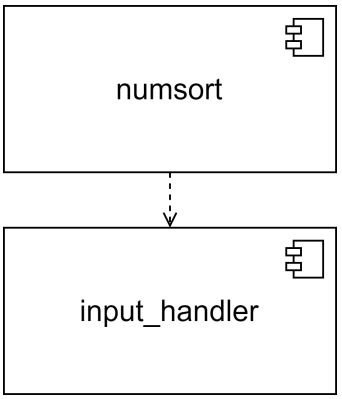

Abbildung 4.16: Building Block View der `REQ-1`, eigene Darstellung

Zu jeder Komponente gibt es eine detaillierte Beschreibung. Die Beschreibung der Komponente `numsort` wird in Abbildung 4.17 veranschaulicht.

```
33 ### `numsort`
34
35 - **Description**:
36 The `numsort.py` script serves as the entry point of the
   application.
37 It initializes the program and orchestrates the execution
   flow by invoking the appropriate modules and functions
   based on user input.
38 - **Responsibilities**:
39   - Defines command-line options.
40   - Parses and validates user input.
41   - Provides help messages.
42   - Handles high-level exceptions.
43   - Initiates the sorting process by calling other modules.
44 - **Interactions**:
45   - Calls functions from `input_handler`.
46   - Passes user-specified parameters to other modules.
47   - Collects data from other components to present results
   or error messages.
```

Abbildung 4.17: Textuelle Beschreibung der Komponente `numsort`, eigene Darstellung

- **Runtime View**: Ein Sequenzdiagramm, das den Ablauf der Anwendung während der Ausführung darstellt. Das entsprechende Diagramm der `REQ-1` wird in Abbildung 4.18 dargestellt.

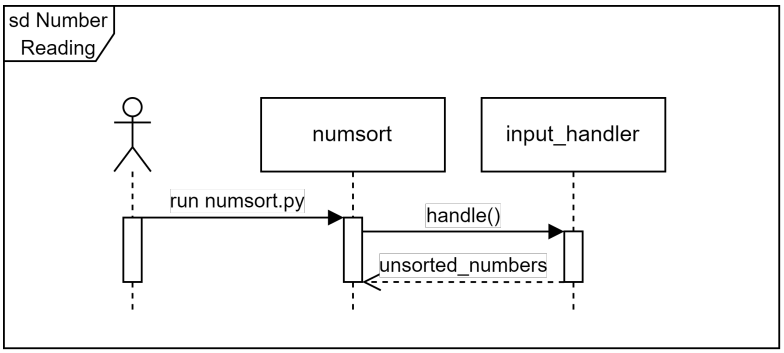

Abbildung 4.18: Runtime View der `REQ-1`, eigene Darstellung

Die textuelle Beschreibung erläutert detailliert, wie die einzelnen Komponenten zusammenwirken, um die Funktionalität der Anwendung bereitzustellen. Dies umfasst den Prozess vom Einlesen der Daten über das Sortieren bis hin zur Ausgabe der

Ergebnisse. Die ersten drei Ablaufschritte werden in Abbildung 4.19 veranschaulicht.

```
100 ### Initialization
101
102 - The user runs `numsort.py` with the required
    command-line arguments.
103 - The `numsort` entry point parses initial arguments and
    invokes other modules based on user input.
104
105 ### Command Parsing
106
107 - The `numsort` module processes command-line inputs using
    Click and validates arguments and options.
108
109 ### Input Handling
110
111 - `numsort` calls `input_handler` with the input file path.
112 - `input_handler`
113   - reads the CSV file,
114   - parses the numbers, and
115   - validates the data.
116 - Handled data is returned to the `numsort` module.
```

Abbildung 4.19: Erste drei Ablaufschritte in der Runtime View der REQ-1, eigene Darstellung

- **Erweiterbarkeit und zukünftige Verbesserungen**: Hier wird beschrieben, wie die Anwendung erweitert werden kann. Es werden Möglichkeiten aufgezeigt, wie neue Sortieralgorithmen hinzugefügt, weitere Eingabe- und Ausgabeformate unterstützt oder zusätzliche Funktionen implementiert werden können. Dieser Abschnitt dient als Leitfaden für zukünftige Entwicklungen und stellt sicher, dass die Anwendung skalierbar und anpassungsfähig bleibt. Er wird in Abbildung 4.20 dargestellt.

```
136 ## Extensibility and Future Enhancements
137
138 The modular architecture of NumSort allows for easy
    addition and modification of components:
139
140 - **Supporting Additional Input Formats**:
141   - Extend `input_handler` to support formats like JSON,
    XML, or databases.
142   - Introduce new parsers and writers while maintaining a
    consistent interface.
143 - **Enhancing Command-Line Interface**:
144   - Add new options in the `numsort` module to provide
    extra functionality.
145   - Implement features like verbose logging, performance
    metrics, or algorithm customization.
```

Abbildung 4.20: Abschnitt „Erweiterbarkeit und zukünftige Verbesserungen“ der Architekturdokumentation der `REQ-1`, eigene Darstellung

Weitere ausführliche Informationen sind in der Datei `NumSort Architecture Documentation.md` enthalten.

Mit diesem Schritt ist die Architektur der Anwendung umfassend dokumentiert, und Entwickler verfügen über eine klare Orientierungshilfe für die Implementierung. Dies erleichtert nicht nur die aktuelle Entwicklung, sondern unterstützt auch zukünftige Erweiterungen und Wartungsarbeiten.

4.2.4 Quellcode-Dokumentation erstellen

Wie es in Abschnitt 3.2.1 beschrieben wurde, gibt es mehrere Ebenen der Quellcode-Dokumentation. Da die Module bereits in der Architekturdokumentation ausführlich beschrieben wurden, wurde auf eine separate Quellcode-Dokumentation für die Module selbst verzichtet. Stattdessen wurden Klassen, Funktionen und bestimmte Code-Anweisungen detailliert dokumentiert.

Der Prozess begann mit dem Modul `numsort`. Da dieses Modul hauptsächlich der Orchestrierung der Gesamtfunktionalität dient, wurde direkt der Einstiegspunkt in das Programm definiert (siehe Abbildung 4.21).

```python
@click.command()
@click.argument(
    "file_path",
    type=click.Path(exists=True, dir_okay=False, readable=True)
)
def numsort_cli(file_path: str):
    """NumSort is a command-line application designed to sort
    lists of numbers using a variety of well-known in-place sorting
    algorithms, such as Bubble Sort. It accepts input in the form
    of CSV (Comma-Separated Values) files and outputs the sorted
    data in the same format. NumSort is ideal for users who need a
    simple and efficient tool to organize numerical data or for
    educational purposes to understand how different sorting
    algorithms work in practice.
    """

    pass


# This is the main entry point for the NumSort application.
if __name__ == "__main__":
    numsort_cli()
```

Abbildung 4.21: Quellcode-Dokumentation für das Modul `numsort` für die erste Anforderung in NumSort, eigene Darstellung

Anschließend wurde mit dem nächsten Modul, `input_handler`, fortgefahren. Hier wurde zuerst die Funktion dokumentiert, die für die Auswahl der passenden Eingabeverarbeitungsfunktion zuständig ist (siehe Abbildung 4.22). Diese Funktion entscheidet basierend auf dem Eingabeformat, welche spezifische Verarbeitungsfunktion verwendet werden soll. Abschließend wurde die spezifische Verarbeitungsfunktion für CSV-Dateien detailliert beschrieben.

```python
def handle(file_path: str) -> list[float]:
    """Selects the correct handler for the file and delegates
    the task to it.

    Args:
        file_path (str): Must be a path to an existing file.

    Returns:
        list: Float numbers from the file.

    Raises:
        ValueError: If the file extension is not supported.
    """

    pass


def handle_csv(file_path: str) -> list[float]:
    """Reads data from a CSV file, validates it, converts it
    to float numbers, and returns the numbers.

    Args:
        file_path (str): Must be a path to an existing CSV
        file.

    Returns:
        list: Float numbers from the file.

    Raises:
        ValueError: If the file is empty or contains invalid
        data.
    """

    pass
```

Abbildung 4.22: Quellcode-Dokumentation für das Modul `input_handler` für die erste Anforderung in NumSort, eigene Darstellung

Die Wahl eines Top-Down-Ansatzes für die Quellcode-Dokumentation wurde bewusst getroffen, da er sich nahtlos in das Laufzeitdiagramm der Architekturdokumentation einfügt. Dieser Ansatz beginnt mit den übergeordneten Modulen, die den Gesamtfluss und die Struktur des Programms definieren, und arbeitet sich schrittweise zu den konkreten

Implementierungsdetails vor. Dadurch wird es möglich, zunächst den Gesamtzusammenhang zu erfassen und dann die einzelnen Funktionen in ihrem jeweiligen Kontext zu verstehen.

Ein Bottom-Up-Ansatz hätte in diesem Fall Schwierigkeiten bereitet, da zunächst unklar gewesen wäre, wie die einzelnen Bausteine optimal in das Gesamtsystem integriert werden können. Ohne eine vorab definierte Gesamtarchitektur hätte sich der Entwicklungsprozess verzetteln können, da es an einem zentralen Rahmen für die Dokumentation des Programmablaufs gefehlt hätte. Der Top-Down-Ansatz ermöglicht es hingegen, systematisch alle notwendigen Funktionen zu identifizieren und diese im Kontext des gesamten Systems zu dokumentieren, was den Entwicklungsprozess deutlich effizienter gestalten soll.

Durch diese Vorgehensweise ist die Quellcode-Dokumentation eng mit der Anforderung, der Architektur und dem Ablauf der Anwendung verknüpft. Entwickler erhalten dadurch nicht nur Einblick in die Funktion einzelner Codebestandteile, sondern auch in deren Rolle innerhalb des Gesamtsystems.

4.2.5 Tests erstellen

Die Erstellung von Tests ist ein wesentlicher Schritt, um sicherzustellen, dass alle Akzeptanzkriterien der definierten Anforderung erfüllt sind. In diesem Beispiel wurden die Tests bewusst manuell erstellt, anstatt sie mit GitHub Copilot zu generieren. So konnte gewährleistet werden, dass sie exakt die in der Dokumentation spezifizierten Anforderungen überprüfen. Dafür wurde für jedes Modul ein eigenes Testmodul entwickelt, das systematisch die korrekte Funktionalität der Implementierung prüft.

Aus der Python-Standardbibliothek wurde das `unittest`-Modul für das Testen der Anwendung ausgewählt. Dieses Modul bietet eine robuste Grundlage für das Schreiben und Ausführen von Tests und unterstützt die Organisation von Tests in übersichtlichen Strukturen.

Die Vorgehensweise bei der Testentwicklung umfasst folgende Schritte:

- **Erstellen von Testmodulen**: Für jedes Modul wird ein Testmodul mit dem Namen `module_name_test.py` erstellt, um eine klare Strukturierung der Tests sicherzustellen.

- **Erstellen einer Testklasse pro Funktion**: Für jede zu testende Funktion wird eine eigene Klasse definiert, die von `unittest.TestCase` erbt. Dies ermöglicht eine klare Trennung der Tests und erleichtert die Wartung und Erweiterung des Testcodes.
- **Definieren von Testmethoden für spezifische Anwendungsfälle**: Innerhalb jeder Testklasse werden Methoden erstellt, die die in dem Anforderungsdokument definierten Use Cases prüfen. Diese Methoden beginnen üblicherweise mit dem Präfix `test_`, damit das `unittest`-Modul sie automatisch erkennen und ausführen kann.

Ein Beispiel für eine solche Testklasse ist in Abbildung 4.23 dargestellt.

```
6   class TestHandleCSV(unittest.TestCase):
7       """Tests for the input_handler.handle_csv function."""
8
9       @patch(
10          "builtins.open",
11          new_callable=mock_open,
12          read_data="1,2,3.3,4,5"
13      )
14      def test_valid_csv_is_handled(self, mock_file):
15          unsorted_numbers = [1, 2, 3.3, 4, 5]
16          file_path = "valid.csv"
17          result = input_handler.handle_csv(file_path)
18          mock_file.assert_called_once_with(file_path, "r")
19          self.assertEqual(result, unsorted_numbers)
20
21      @patch(
22          "builtins.open",
23          new_callable=mock_open,
24          read_data=""
25      )
26      def test_empty_file_raises_error(self, _):
27          file_path = "empty.csv"
28          with self.assertRaises(ValueError) as context:
29              input_handler.handle_csv(file_path)
30          self.assertEqual(
31              str(context.exception),
32              f"Error: File '{file_path}' is empty."
33          )
34
35      @patch(
36          "builtins.open",
37          new_callable=mock_open,
38          read_data="1,2,three,4,5"
39      )
40      def test_invalid_data_raises_error(self, _):
41          file_path = "invalid_data.csv"
42          with self.assertRaises(ValueError) as context:
43              input_handler.handle_csv(file_path)
44          self.assertEqual(
45              str(context.exception),
46              f"Error: File '{file_path}' contains invalid data. " +
47              "Expected numbers separated by commas.",
48          )
```

Abbildung 4.23: Tests für Funktion `handle_csv` in NumSort, eigene Darstellung

Diese Klasse testet die Funktion zum Einlesen von Zahlen aus einer CSV-Datei und umfasst Tests für verschiedene Szenarien, einschließlich:

- **Erfolgreiches Einlesen gültiger Daten**: Überprüfung, ob die Funktion eine Liste von Zahlen zurückgibt, wenn eine korrekt formatierte CSV-Datei bereitgestellt wird.
- **Verarbeitung einer leeren Datei**: Testen, ob die Funktion angemessen reagiert, wenn die Datei keine Daten enthält, und entsprechende Maßnahmen ergreift.
- **Umgang mit ungültigen Daten**: Überprüfung, ob die Funktion Fehler erkennt, wenn die Datei nicht-numerische Werte oder ein falsches Format enthält, und entsprechende Fehlerbehandlungen durchführt.

Durch diese strukturierte Testentwicklung wird gewährleistet, dass die Anwendung robust ist und sich in verschiedenen Situationen erwartungsgemäß verhält. Die Tests können regelmäßig ausgeführt werden, um die Integrität des Codes während der Entwicklung zu überprüfen und Regressionen frühzeitig zu erkennen.

4.2.6 Anforderung implementieren

Nachdem ausführlich dokumentiert wurde, was umgesetzt werden soll, erfolgt nun die eigentliche Implementierung der Anforderung. Im Gegensatz zur Dokumentation wird dabei ein Bottom-up-Ansatz gewählt, der es ermöglicht, dass jede neu implementierte Funktionalität auf bereits vorhandene Komponenten aufbaut. Parallel zur Umsetzung werden kontinuierlich Tests durchgeführt, um zu verifizieren, dass die Implementierung den festgelegten Anforderungen entspricht.

Darüber hinaus wird das in Abschnitt 3.2.1 vorgestellte Verwendung der Dokumentation als Kontext für KI-Systeme angewendet. Laut GitHub [Git24] existieren zwei primäre Methoden zur Interaktion mit GitHub Copilot:

1. Copilot Chat.
2. Copilot Code Completion.

Beide Varianten können je nach Größe der zu implementierenden Funktion und dem benötigten Kontext effektiv eingesetzt werden. Während Copilot Chat hilfreich ist, um größere Codeteile zu generieren oder komplexe Fragen zu beantworten, eignet sich die Code Completion-Funktion hervorragend für kleinere Ergänzungen und schnelle Vervollständigungen. Durch die Kombination dieser Werkzeuge wurde die Anforderung vollständig implementiert.

Copilot Chat

Copilot Chat funktioniert ähnlich wie andere KI-basierte Chat-Tools wie ChatGPT. Es ermöglicht eine interaktive Kommunikation mit der KI, um Fragen zu stellen oder spezifische Codeabschnitte zu generieren. Laut GitHub [Git24] eignet sich Copilot Chat besonders für folgende Szenarien:

- **Beantwortung von Fragen zum Code in natürlicher Sprache**: Entwickler können Fragen zu bestehenden Codeabschnitten stellen und erhalten erklärende Antworten.
- **Generierung großer Codeabschnitte**: Es unterstützt bei der Erstellung umfangreicherer Codesegmente, die anschließend an individuelle Bedürfnisse angepasst werden können.
- **Ausführung spezifischer Aufgaben mit Schlüsselwörtern und Fähigkeiten**: Copilot Chat verfügt über integrierte Keywords und Skills, die helfen, den Kontext für Prompts bereitzustellen und häufige Aufgaben schnell zu erledigen.
- **Arbeiten aus der Perspektive einer bestimmten Persona**: Man kann Copilot Chat anweisen, als bestimmte Rolle zu agieren.

Ein Beispiel für die Verwendung von Copilot Chat ist in Abbildung 4.24 dargestellt.

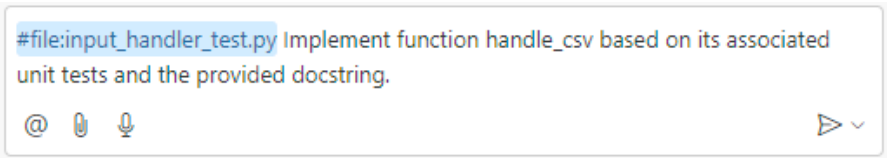

Abbildung 4.24: Prompt für Copilot Chat zur Implementierung der ersten Anforderung in NumSort, eigene Darstellung

Zu Beginn der Nutzung wurde den Kontext definiert, auf den GitHub Copilot zugreifen soll. Dies wurde durch die Auswahl des relevanten Codes in der Datei `input_handler.py` und die Verwendung der Chat-Variable `#file` erreicht. Obwohl Copilot im Hintergrund bereits den umgebenden Code analysiert, kann durch explizite Eingaben sichergestellt werden, dass genau der gewünschte Kontext berücksichtigt wird.

Nachdem der Kontext festgelegt war, wurde beschrieben, welche Funktionalität GitHub Copilot generieren soll. Basierend auf diesen Eingaben hat Copilot einen passenden Vorschlag erstellt, der den Anforderungen entsprach (siehe Abbildung 4.25).

GitHub Copilot

Used 2 references

input_handler_test.py NumSort • source
input_handler.py:4-31 NumSort • source

Plan

1. Import the necessary `csv` module.
2. Open the file specified by `file_path`.
3. Read the contents of the file using `csv.reader`.
4. Check if the file is empty or contains no data.
5. Convert the data to a list of floats.
6. Handle any exceptions that occur during the conversion.
7. Return the list of floats.

Implementation

```
import csv

def handle_csv(file_path: str) -> list[float]:
    """Reads data from a CSV file, validates it, converts it to

    Args:
        file_path (str): Must be a path to an existing CSV file

    Returns:
        list: Float numbers from the file.

    Raises:
        ValueError: If the file is empty or contains invalid dat
    """
    with open(file_path, "r") as file:
        reader = csv.reader(file)
        data = list(reader)

        if not data or not data[0]:
            raise ValueError(f"Error: File '{file_path}' is empt

        try:
            numbers = [float(item) for item in data[0]]
        except ValueError:
            raise ValueError(
                f"Error: File '{file_path}' contains invalid dat
            )

        return numbers
```

Abbildung 4.25: Ergebnis der Copilot Chat-Interaktion zur Implementierung der ersten Anforderung in NumSort, eigene Darstellung

Der generierte Code wurde anschließend in das entsprechende Modul eingefügt und an die spezifischen Anforderungen der Anwendung angepasst.

Copilot Code Completion

Die Code Completion-Funktion von GitHub Copilot bietet automatisierte Codevorschläge während des Schreibens. Laut GitHub [Git24] ist diese Methode besonders effektiv für:

- **Vervollständigen von Codefragmenten, Variablennamen und Funktionen**: Während man Code schreibt, schlägt Copilot mögliche Fortsetzungen vor.
- **Generierung von repetitivem Code**: Hilfreich bei sich wiederholenden Strukturen oder Mustern im Code.
- **Erstellung von Code aus Inline-Kommentaren in natürlicher Sprache**: Durch das Schreiben von Kommentaren kann Copilot entsprechenden Code generieren.
- **Generierung von Tests für testgetriebene Entwicklung**: Unterstützung beim Schreiben von Unit-Tests basierend auf vorhandenen Funktionen.

Ein Beispiel für einen Code Completion-Vorschlag ist in Abbildung 4.26 dargestellt.

```
34  def handle(file_path: str) -> list[float]:
35      """Selects the correct handler for the file and delegates the task to it.
36
37      Args:
38          file_path (str): Must be a path to an existing file.
39
40      Returns:
41          list: Float numbers from the file.
42
43      Raises:
44          ValueError: If the file extension is not supported.
45      """
46
47      if file_path.endswith(".csv"):
            return handle_csv(file_path)

        raise ValueError(f"Error: File extension is not supported. Supported extensions: '.csv'.")
```

Abbildung 4.26: Verwendung von Copilot Code Completion zur Implementierung der `REQ-1`, eigene Darstellung

Hier schlägt Copilot basierend auf der bereits geschriebenen Quellcode-Dokumentation und dem Kontext passende Codezeilen vor, die direkt übernommen oder angepasst werden können.

4.2.7 Bereitstellung

Nach erfolgreicher Implementierung der `REQ-1` wurde auf GitHub ein Release mit dem Tag `v0.1.0` erstellt (siehe Abbildung 4.27). Diese Version entspricht der in der Benutzerdokumentation angegebenen Version, sodass Benutzer überprüfen können, ob sie die aktuelle Version der Dokumentation verwenden.

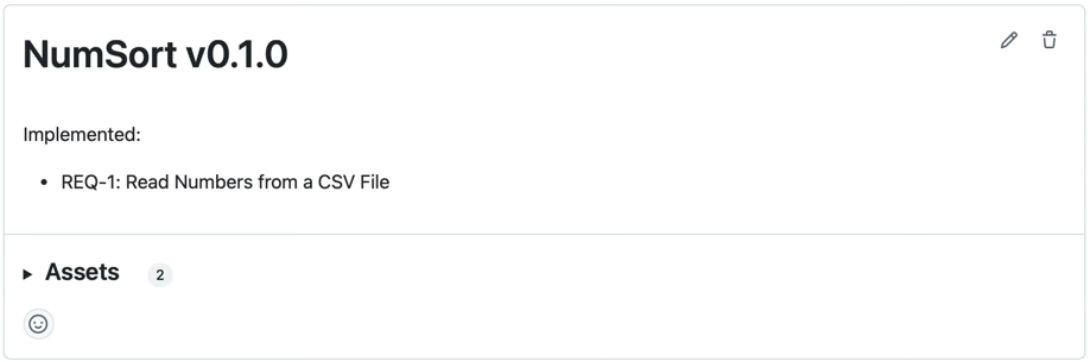

Abbildung 4.27: Release von NumSort v0.1.0 auf GitHub, eigene Darstellung

In den Release-Informationen wurde deutlich gemacht, dass die erste Anforderung, nämlich das Einlesen von Zahlen aus einer CSV-Datei, erfolgreich umgesetzt wurde. Alle Dateien des Repositories wurden als Assets dem Release hinzugefügt. Dazu gehören die Software-Dokumentation und die ausführbare Dateien der Anwendung.

Durch die Bereitstellung dieses Releases wird sichergestellt, dass Benutzer und Entwickler auf eine konsistente und geprüfte Version der Anwendung zugreifen können.

5 Evaluation des KI-gestützten DDD anhand des Proof of Concept

In folgenden Abschnitten wird kritisch analysiert, welchen praktischen Einfluss KI-gestütztes DDD auf die in Abschnitt 2.3 beschriebenen Herausforderungen der Software-Dokumentation hat. Diese Analyse soll nicht nur die Vorteile hervorheben, sondern auch potenzielle Herausforderungen und Grenzen des Ansatzes aufzeigen.

5.1 Umfang und Detaillierungsgrad der Dokumentation

Während der Umsetzung des Proof of Concept stand der Autor unter dem Druck, möglichst schnell mit der tatsächlichen Implementierung zu beginnen. Dies spiegelt eine häufige Situation in der realen Softwareentwicklung wider, in der enge Fristen und Termindruck die verfügbare Zeit für ausführliche Dokumentation einschränken. Aus diesem Grund wurde bewusst entschieden, nur so viel Dokumentation zu erstellen, wie es für die beteiligten Stakeholder tatsächlich erforderlich war. Dabei wurde versucht, einen Mittelweg zu finden: Die Dokumentation sollte klar und verständlich sein, ohne jedoch unnötig ins Detail zu gehen.

Da das Proof of Concept von nur einer Person erstellt wurde, besteht die Möglichkeit, dass der Umfang und der Detaillierungsgrad der Dokumentation nicht vollständig den Bedürfnissen aller potenziellen Stakeholder entsprechen. In einem realen Projektumfeld wäre es essenziell, durch kontinuierliche Kommunikation mit den Stakeholdern deren Anforderungen an die Dokumentation zu ermitteln und entsprechend anzupassen. Dies entspricht den Prinzipien agiler Methoden, bei denen die Zusammenarbeit mit dem Kunden und die Reaktion auf Veränderungen über umfangreiche Dokumentation gestellt werden.

Das KI-gestützte DDD hat zusätzlich dazu beigetragen, mögliche Wiederholungen in der Dokumentation zu vermeiden. Durch eine sorgfältig durchdachte Struktur der Soft-

waredokumentation, die bereits in Szenario 1 (siehe Abschnitt 4.1.3) festgelegt wurde, war von Anfang an klar, welche Arten von Dokumentation verwendet werden und welche Informationen in welchen Dateien enthalten sein sollen. Dies entspricht dem Prinzip der DRY-Methode, bei dem Redundanzen vermieden werden, um die Wartbarkeit und Konsistenz der Dokumentation zu erhöhen.

Insgesamt bestätigten die Erfahrungen aus dem Proof of Concept die in Abschnitt 2.3.1 beschriebenen Beobachtungen.

5.2 Dokumentationsschuld

Der zentrale Ansatz des KI-gestützten DDD besteht darin, die Dokumentation vor der eigentlichen Implementierung zu erstellen. Durch dieses Vorgehen soll sichergestellt werden, dass der Code direkt auf der Dokumentation basiert und beide stets synchron sind. In diesem Kontext erscheint es zunächst unwahrscheinlich, dass eine Dokumentationsschuld - also die Diskrepanz zwischen veralteter Dokumentation und aktuellem Code - entsteht.

Dennoch gibt es Situationen, in denen Dokumentationsschuld auftreten kann. Eine besonders kritische Phase ist gegeben, wenn nachträglich umfangreiche Änderungen am Quellcode vorgenommen werden müssen. Wenn Entwickler beispielsweise die Quellcode-Dokumentation eines großen Codeabschnitts ändern, um die Logik zu aktualisieren, besteht die Gefahr, dass sie vergessen, den Code entsprechend anzupassen. Dieses Risiko ist besonders hoch, wenn für die vorgenommenen Änderungen keine automatisierten Tests geschrieben wurden, was aus Zeitgründen oder aufgrund von Komplexität nicht immer möglich ist. Ohne ausreichende Tests können Inkonsistenzen zwischen Dokumentation und Code unentdeckt bleiben, was zu Fehlern und Missverständnissen führt.

Ein möglicher Ansatz zur Minimierung dieses Risikos ist die Verwendung von `TODO`-Kommentaren im Code. Entwickler können an Stellen, die einer Überarbeitung bedürfen, entsprechende Markierungen setzen. Beim späteren Durcharbeiten des Codes können diese Kommentare gezielt angesprochen und die notwendigen Anpassungen vorgenommen werden [OBr+24]. Tools zur statischen Codeanalyse oder Integrated Development Environments (IDEs) unterstützen häufig die Verwaltung von `TODO`-Kommentaren und erleichtern so die Nachverfolgung offener Aufgaben.

Allerdings löst das KI-gestützte DDD nicht automatisch die bereits bestehende Dokumentationsschuld in einem bestehenden Projekt. Wenn dieser Ansatz in ein vorhandenes System integriert wird, bleibt die bereits angesammelte Dokumentationsschuld bestehen. Es gilt zu untersuchen, wie bestehende Dokumentationsschuld effektiv identifiziert und behoben werden kann, insbesondere im Zusammenhang mit der Integration von KI-gestütztem DDD. Methoden wie automatisierte Dokumentationsgenerierung aus dem Code könnten hierbei hilfreich sein [RBG22; WZW21].

Insgesamt bestätigen die Erfahrungen aus dem Proof of Concept die in Abschnitt 2.3.2 beschriebenen Beobachtungen. Trotz eines dokumentationsgetriebenen Ansatzes kann Dokumentationsschuld entstehen, insbesondere wenn Änderungen nicht konsequent in allen betroffenen Artefakten nachvollzogen werden. Das KI-gestützte DDD reduziert zwar das Risiko der Entstehung von Dokumentationsschuld, eliminiert es aber nicht vollständig. Es bleibt daher essenziell, Prozesse zur kontinuierlichen Pflege und Aktualisierung der Dokumentation zu etablieren und das Bewusstsein der Entwickler für dieses Thema zu schärfen.

5.3 Kosten-Nutzen-Abwägung

Während der Arbeit an NumSort wurde die aufgewendete Zeit für verschiedene Entwicklungsaktivitäten gemessen, um einen Einblick in die Kosten-Nutzen-Abwägung des KI-gestützten DDD zu erhalten. Die Messung erfolgte unter den folgenden Annahmen:

- Der Autor war der alleinige Entwickler des Projekts (weiter unten als „der Entwickler“ bezeichnet).
- Der Entwickler hatte keine berufliche Erfahrung mit den verwendeten Technologien, abgesehen von einigen Studienprojekten mit Python, Markdown und GitHub Copilot.
- Die Komplexität der Anforderungen `REQ-1`, `REQ-2` und `REQ-3` war ungefähr gleich, wobei dem Entwickler der Sortieralgorithmus bereits bekannt war.
- Es wurde nur die Zeit gemessen, die für die relevanten Aktivitäten direkt im Zusammenhang mit der Entwicklung stand.
- Die Ergebnisse wurden gerundet und in Minuten angegeben.

Das Ziel dieser Messung war es, einen groben Überblick darüber zu erhalten, wie sich die Zeiten für das Schreiben von Dokumentation, Tests und Code in diesem spezifischen Beispiel unterscheiden und wie sie sich mit weiteren Iterationen verändern. Die Ergebnisse sind in Abbildung 5.1 dargestellt.

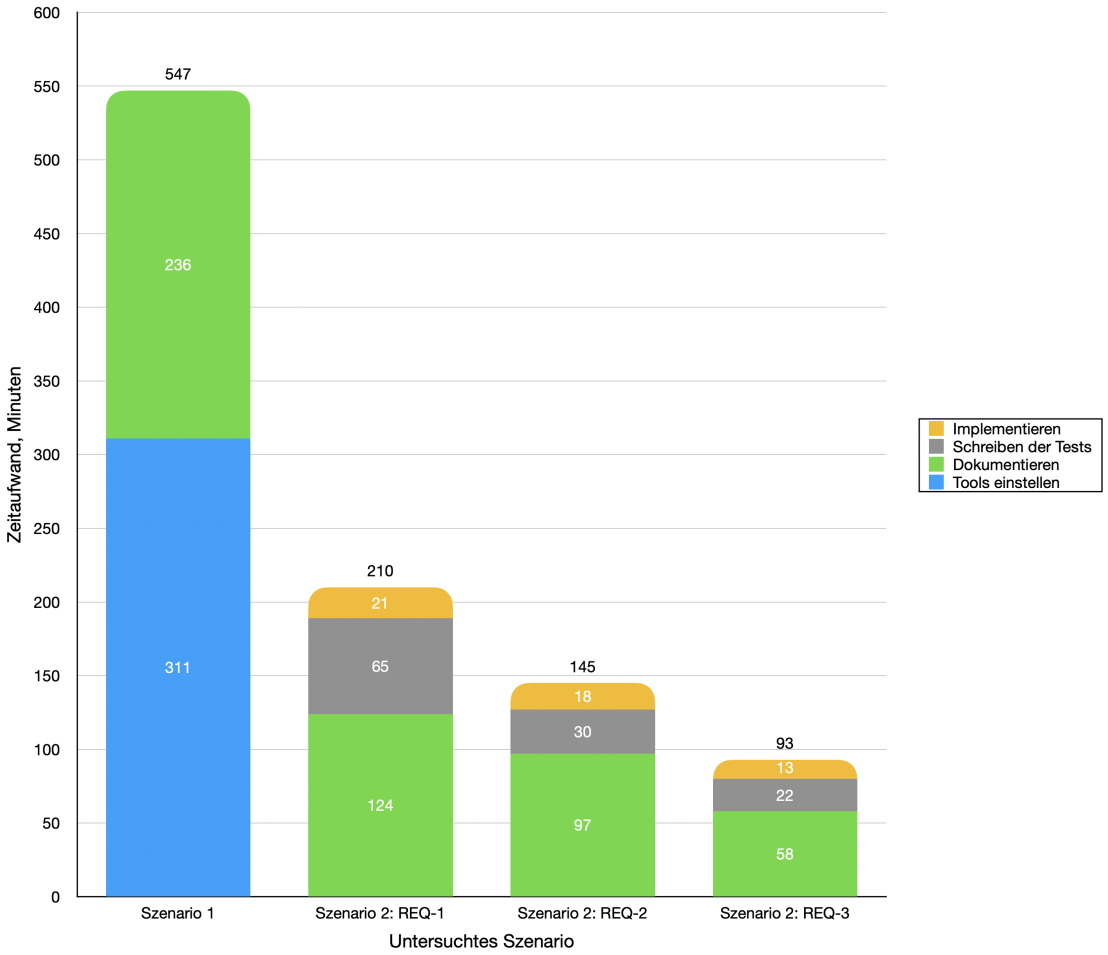

Abbildung 5.1: Zeitmessung für Proof of Concept (in Minuten), eigene Darstellung

Szenario 1 erforderte mehr Zeit als die nachfolgenden Szenarien. Dies lag hauptsächlich daran, dass es notwendig war, das Projekt aufzusetzen, sich mit den Tools vertraut zu machen, die Softwaredokumentation zu organisieren und die initiale Dokumentation zu erstellen. Diese Vorarbeiten sind essentiell, um eine solide Basis für die weitere Entwicklung zu schaffen. Insbesondere die Prozessdokumentation erwies sich als hilfreich, da sie eine klare Anleitung für den Entwicklungsprozess lieferte. Durch eine nachvollziehbare

Vorgehensweise konnte der Entwickler sich auf die eigentlichen Entwicklungsaktivitäten konzentrieren, ohne von organisatorischen Fragen abgelenkt zu werden.

In Szenario 2 zeigte sich, dass der zusätzliche Aufwand für Dokumentation und Tests bei jeder neuen Anforderung signifikant war. Bevor mit der eigentlichen Implementierung begonnen werden konnte, musste der Entwickler sich ausführlich mit der Erstellung der Dokumentation und der Tests beschäftigen. Dieser Ansatz entspricht den Prinzipien des DDD, bei dem die Dokumentation die Grundlage für die Entwicklung bildet.

Dank GitHub Copilot verlief das Generieren des Codes sehr effizient. Bei der Implementierung des Sortieralgorithmus, der ein standardisiertes Verfahren darstellt, konnte Copilot auf umfangreiche Trainingsdaten zurückgreifen und entsprechend passende Codevorschläge machen. Durch die Verwendung der Dokumentation als Kontext (siehe Abschnitt 3.2.1) wurden die generierten Codeabschnitte weitgehend an die Anforderungen angepasst und erforderten nur minimale Änderungen.

Allerdings wurde bei `REQ-2` (Implementierung des Bubble Sort-Algorithmus) festgestellt, dass Copilot eine nicht optimierte Version des Algorithmus vorschlug. Die von GitHub Copilot generierte Version implementierte den Bubble Sort-Algorithmus ohne vorzeitige Abbruchmöglichkeit. In einer optimierten Version lässt sich der Algorithmus durch Einführung einer Abbruchbedingung verbessern: Wenn in einem Durchlauf kein Tauschvorgang stattfindet, bedeutet dies, dass die Zahlen bereits vollständig sortiert sind und weitere Durchläufe überflüssig sind [fre19]. Das Problem kann auch in der zugehörigen Quellcode-Dokumentation liegen, da nicht explizit erwähnt wurde, dass der Bubble Sort-Algorithmus mit dieser Optimierung implementiert werden muss. Obwohl diese Version des Algorithmus weit verbreitet und als Standard vorausgesetzt wird [fre19], führt das Fehlen eines klaren Hinweises leicht zu einer nicht optimierten Implementierung. Dies verdeutlicht, dass eine manuelle Überprüfung des generierten Codes weiterhin unerlässlich ist. Es unterstreicht die Notwendigkeit, die Vorschläge der KI kritisch zu hinterfragen und sie oder die entsprechende Dokumentation gegebenenfalls gezielt anzupassen.

Außerdem konnte es nicht eindeutig festgestellt werden, inwieweit das KI-gestützte DDD das Schreiben des Codes insgesamt beschleunigt hat. Die gemessenen Zeiten zeigen zwar, dass das eigentliche Coding weniger Zeit in Anspruch nahm als Dokumentation und Tests, doch ein direkter Vergleich mit einem traditionellen Entwicklungsansatz ohne KI-Unterstützung fehlt. Um belastbare Aussagen treffen zu können, wären weitere Forschungen notwendig.

Darüber hinaus zeigt die Zeitmessung deutlich, dass die Erstellung von Dokumentation und Tests bei jeder Anforderung den größten Aufwand verursacht. Diese Tatsache lässt sich noch anschaulicher in der prozentualen Darstellung nachvollziehen (siehe Abbildung 5.2).

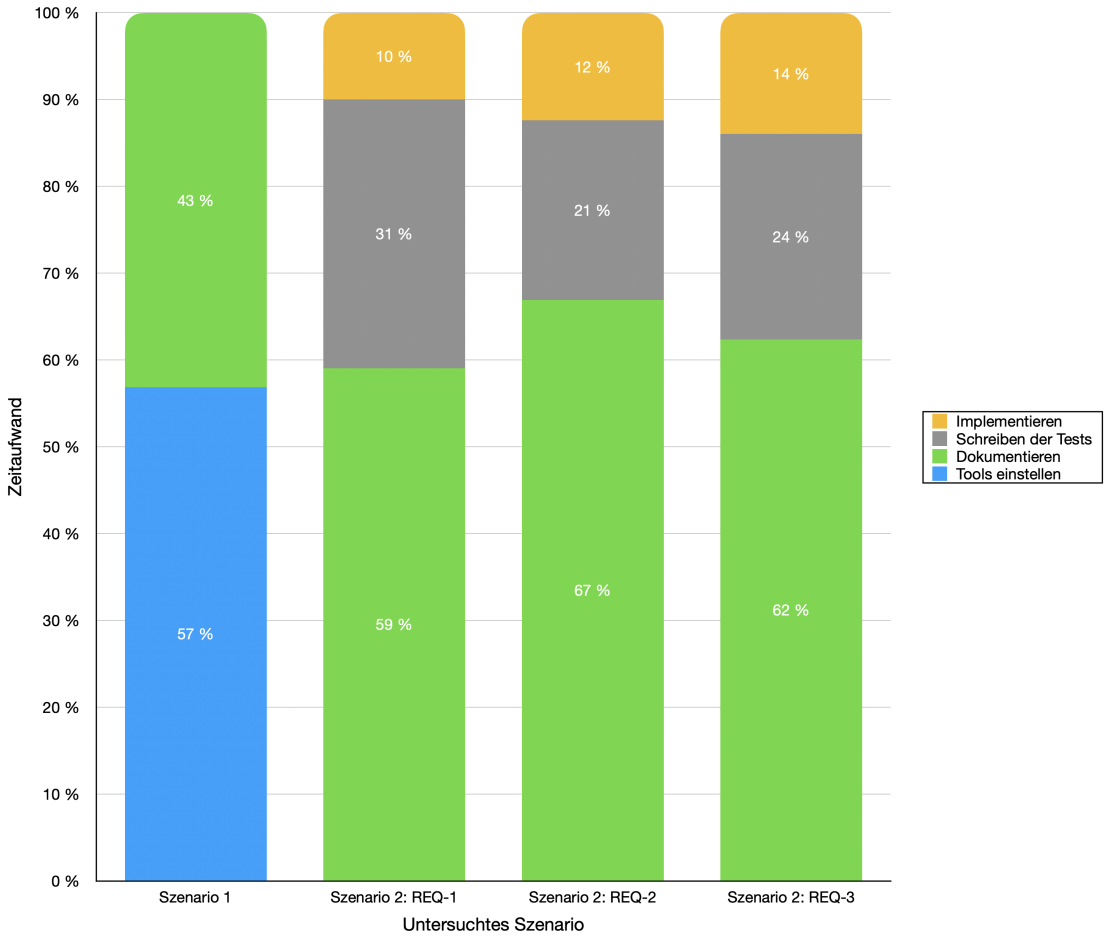

Abbildung 5.2: Zeitmessung für Proof of Concept (prozentual), eigene Darstellung

Um diesen Prozess effizienter zu gestalten, können verschiedene Ansätze verfolgt werden.

Eine Möglichkeit ist die automatisierte Dokumentationserstellung mit Hilfe von KI. Hierbei könnten Gerüste und Vorlagen bereitgestellt werden, um der KI bei der Generierung einer standardisierten Dokumentation zu helfen. Entwickler skizzieren die wesentlichen Inhalte der Dokumente und liefern der KI ein Beispiel für das gewünschte Endergebnis, wodurch sie Zeit beim Formatieren und Schreiben wiederkehrender Textbausteine

sparen. Allerdings muss die generierte Dokumentation sorgfältig geprüft werden, um sicherzustellen, dass sie korrekt und vollständig ist. Gegebenenfalls sind logische Fehler oder Missverständnisse der KI zu korrigieren.

Ein weiterer Ansatz ist die automatisierte Testgenerierung basierend auf der Anforderungsdokumentation. Die in den Anforderungen beschriebenen Use Cases können als Grundlage für die automatische Erstellung von Tests dienen. Auch der Einsatz von GitHub Copilot hat sich als hilfreich erwiesen, da er sinnvolle Vorschläge für Testfälle machen kann, selbst wenn keine genauen Anweisungen vorliegen, welchen Use Cases die Tests entsprechen sollen. Durch die gezielte Nutzung von KI-Unterstützung bei der Testentwicklung lässt sich die Effizienz steigern, was zu einer deutlichen Zeitersparnis führt. Auch hier ist es jedoch wichtig, die automatisch generierten Tests genau zu überprüfen, um sicherzustellen, dass sie die Anforderungen vollständig abdecken und korrekt implementiert sind.

Insgesamt bestätigen die Erfahrungen aus dem Proof of Concept die in Abschnitt 2.3.3 beschriebenen Beobachtungen. Während das KI-gestützte DDD Potenziale zur Effizienzsteigerung bietet, insbesondere beim Schreiben von Code, bleibt der zusätzliche Aufwand für Dokumentation und Tests bestehen. Es ist daher nicht eindeutig, ob das KI-gestützte DDD den gesamten Overhead des traditionellen DDD reduzieren kann.

5.4 Qualität der Dokumentation

Wie in Abschnitt 2.3.4 beschrieben, fördert das KI-gestützte DDD eine Verbesserung der Qualität der Dokumentation, da diese mit besonderer Sorgfalt und Aufmerksamkeit erstellt wird. Die Vorverlagerung der Dokumentation in den Entwicklungsprozess stellt sicher, dass Anforderungen klar definiert und verständlich kommuniziert werden, was wiederum die Grundlage für qualitativ hochwertigen Code bildet.

Im Rahmen des durchgeführten Proof of Concept war es jedoch schwierig, eine objektive Bewertung der erstellten Dokumentation vorzunehmen. Einerseits war das Ziel des Projekts nicht die Entwicklung einer vollständig ausgereiften und perfekt dokumentierten Anwendung, sondern die Demonstration der Prinzipien des KI-gestützten DDD. Andererseits fehlten externe Stakeholder oder Benutzer, deren Feedback zur Qualität der Dokumentation hätte herangezogen werden können. Ohne solche Rückmeldungen bleibt die Beurteilung der Dokumentationsqualität subjektiv und auf die eigene Wahrnehmung

beschränkt. Daher besteht hier Bedarf für weitere Forschung und praktische Anwendungen in realen Projekten, um belastbare Aussagen über die Qualitätsverbesserungen durch KI-gestütztes DDD treffen zu können.

Ein bemerkenswerter Aspekt war die Beobachtung, dass GitHub Copilot am effektivsten mit textuellen Darstellungen von Informationen arbeitet. Während textbasierte Dokumentationen und Kommentare gut von der KI verarbeitet und für die Codegenerierung genutzt werden können, besteht eine Herausforderung bei der Einbindung von Diagrammen und visuellen Modellen. Da Diagramme in der Regel als Bilddateien oder proprietäre Formate vorliegen, kann GitHub Copilot diese nicht immer richtig interpretieren oder darauf basierende sinnvolle Codevorschläge generieren. Sie können aber die Qualität der Dokumentation verbessern, indem sie komplexe Zusammenhänge und Strukturen veranschaulichen.

Um dieses Problem zu umgehen, ist es sinnvoll, zusätzliche textuelle Beschreibungen für alle Diagramme zu erstellen. Diese Beschreibungen sollten die wesentlichen Inhalte und Zusammenhänge der Diagramme erläutern. Wenn es gewünscht ist, dass GitHub Copilot auf diese Informationen zugreift, können die Beschreibungen als Kommentare oder separate Textdokumente eingebunden werden. Dies erhöht jedoch den Aufwand für die Dokumentation und kann zu Redundanzen führen.

Ein weiteres Problem stellt die Skalierbarkeit der erstellten Diagramme mit zunehmender Anwendungsgröße dar. In kleinen Projekten sind Diagramme übersichtlich und leicht verständlich. Bei größeren Systemen können sie jedoch schnell unübersichtlich werden. Eine mögliche Lösung besteht darin, die Diagramme in verschiedene Sichten oder Ebenen zu unterteilen. Jede Sicht fokussiert auf bestimmte Aspekte des Systems und bleibt dadurch handhabbar. Allerdings würde eine textuelle Beschreibung jeder Sicht zusätzlichen Aufwand bedeuten.

Mit der Weiterentwicklung von GitHub Copilot und ähnlichen KI-Tools könnte sich diese Situation jedoch ändern. Zukünftige Versionen könnten in der Lage sein, visuelle Informationen besser zu verarbeiten oder integrative Lösungen anbieten, die Diagramme direkt interpretieren können. Forschungsarbeiten im Bereich der Bildverarbeitung und des maschinellen Sehens könnten hierzu beitragen.

Zusammenfassend lässt sich sagen, dass das KI-gestützte DDD das Potenzial hat, die Qualität der Dokumentation zu verbessern, indem es den Fokus auf eine sorgfältige und durchdachte Dokumentation legt. Gleichzeitig bestehen jedoch noch Herausforderungen,

insbesondere in Bezug auf die Integration nicht-textueller Informationen und die Skalierbarkeit in größeren Projekten. Weitere Untersuchungen und praktische Erfahrungen sind notwendig, um diese Herausforderungen zu adressieren und das volle Potenzial von KI-gestütztem DDD auszuschöpfen.

6 Fazit und Ausblick

Es ist praktisch unmöglich, zunächst Code zu schreiben und erst danach zu überlegen, welche Funktionalität tatsächlich implementiert werden soll. Durch das KI-gestützte DDD können Entwickler ihre Ideen und Konzepte von Anfang an in Form von Dokumentation festhalten und gemeinsam diskutieren. KI-Tools wie GitHub Copilot können diese dokumentierten Ideen anschließend in Code umsetzen, wobei der Entwickler als kontrollierende Instanz agiert und die Qualität sowie die Übereinstimmung mit den Anforderungen sicherstellt.

Die Beobachtungen von Forschern und Entwicklern (siehe Abschnitt 2.3) zeigen deutlich, dass die Integration von KI in DDD ein großes Potenzial besitzt. Bei der Evaluation des im Rahmen dieser Arbeit entwickelten Proof of Concept (siehe Abschnitt 5) wurden zahlreiche konkrete Vorteile identifiziert:

- Die klar definierte Prozessdokumentation bietet eine verlässliche Anleitung für den Entwicklungsprozess und erlaubt es, einen angemessenen Detaillierungsgrad für die gesamte Dokumentation zu finden, der den tatsächlichen Anforderungen der Stakeholder entspricht.
- Durch das Erstellen der Dokumentation vor der Implementierung wird das Risiko von Dokumentationsschuld reduziert, da Code und Dokumentation von Beginn an synchronisiert sind.
- Das KI-gestützte DDD ermöglicht eine Codegenerierung basierend auf der vorhandenen Dokumentation, wobei GitHub Copilot passende Codevorschläge liefert, die nur minimale Anpassungen erfordern.
- Die sorgfältig strukturierte Dokumentation verhindert Redundanzen und verbessert somit Wartbarkeit und Konsistenz.

Insbesondere kann der Aufwand für das Schreiben und die Pflege von Dokumentation reduziert oder zumindest ausgeglichen werden. Dies ermöglicht es Entwicklern, eine natürliche Denkweise während des Softwareentwicklungsprozesses zu verfolgen, bei der die Implementierung im Voraus durchdacht wird.

Trotz der vielversprechenden Ergebnisse besteht ein hoher Bedarf an weiterer Forschung in diesem Bereich. Während der Recherche wurde festgestellt, dass es nur wenige wissenschaftliche Arbeiten zum Thema DDD gibt. Zukünftige Fortschritte im Bereich der KI könnten die Situation weiter verbessern und neue Möglichkeiten für die effiziente Integration in den Entwicklungsprozess eröffnen.

Ein mögliches Forschungsvorhaben könnte untersuchen, ob das Schreiben der Dokumentation vor dem Code tatsächlich zeitintensiver ist als der umgekehrte Ansatz. Dabei sollte jedoch berücksichtigt werden, dass eine nachträgliche Dokumentation des Codes ebenfalls zeitaufwändig ist und möglicherweise zu einer geringeren Dokumentationsqualität führt. Zwei Gruppen von Entwicklern könnten gebildet werden, um eine Anwendung anhand einer vorgegebenen Anforderungsliste zu implementieren. Die eine Gruppe verwendet das KI-gestützte DDD, die andere das traditionelle DDD ohne KI-Unterstützung. Jeder Entwickler arbeitet individuell, und die aufgewendeten Zeiten sowie die Qualität der Ergebnisse werden gemessen und verglichen. Eine solche Studie könnte Aufschluss darüber geben, ob und in welchem Umfang das KI-gestützte DDD Effizienzgewinne bringt.

Darüber hinaus könnten zukünftige Forschungen konkrete Verbesserungsansätze des KI-gestützten DDD untersuchen, die sich aus der Evaluation ergeben haben. Zu diesen zählen:

- Verbesserte KI-gestützte Methoden zur Identifikation und Behebung bestehender Dokumentationsschuld.
- Die automatisierte Erstellung standardisierter Dokumentation durch KI-unterstützte Gerüste und Vorlagen.
- Die automatisierte Testgenerierung auf Basis der Anforderungsdokumentation.
- Integration textbasierter Beschreibungen für Diagramme, um die Verarbeitung visueller Informationen durch KI-Tools zu verbessern.

Besonders die Herausforderung der Skalierbarkeit von Dokumentation in größeren Projekten und die effiziente Vermeidung von Redundanzen unter Einhaltung des DRY-Prinzips verdienen weitere wissenschaftliche Betrachtung. Diese Weiterentwicklungen könnten es

ermöglichen, die Balance zwischen Dokumentationsumfang und Entwicklungseffizienz weiter zu verbessern.

Es ist zu hoffen, dass diese Arbeit als Anstoß für weitere Forschungen dient und dazu beiträgt, die Softwareentwicklung in Zukunft effizienter und verständlicher zu gestalten. Die Kombination von DDD mit KI-gestützten Tools bietet ein vielversprechendes Feld, das das Potenzial hat, etablierte Praktiken in der Softwareentwicklung zu überdenken und die Qualität sowie die Wartbarkeit von Softwareprojekten nachhaltig zu verbessern.

Literatur

[Agh+19] Emad Aghajani u. a. „Software Documentation Issues Unveiled“. en. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, Mai 2019, S. 1199–1210. ISBN: 978-1-72810-869-8. DOI: 10.1109/ICSE.2019.00122. URL: https://ieeexplore.ieee.org/document/8811931/.

[Bec03] Kent Beck. *Test-driven development: by example*. The Addison-Wesley signature series. Boston: Addison-Wesley, 2003. 220 S. ISBN: 978-0-321-14653-3.

[Beh+20] Woubshet Behutiye u. a. „How agile software development practitioners perceive the need for documenting quality requirements: a multiple case study“. en. In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Portoroz, Slovenia: IEEE, Aug. 2020, S. 93–100. ISBN: 978-1-72819-532-2. DOI: 10.1109/SEAA51224.2020.00025. URL: https://ieeexplore.ieee.org/document/9226305/.

[Che+23] Leon Chemnitz u. a. *Towards Code Generation from BDD Test Case Specifications: A Vision*. en. arXiv:2305.11619 [cs]. Mai 2023. URL: http://arxiv.org/abs/2305.11619.

[Cho15] Vikas S Chomal. „Software Project Documentation - An Essence of Software Development“. en. In: 06.06 (2015).

[Cla+24] Autumn Clark u. a. „A Quantitative Analysis of Quality and Consistency in AI-generated Code“. In: *2024 7th International Conference on Software and System Engineering (ICoSSE)*. IEEE, Apr. 2024, S. 37–41. DOI: 10.1109/ICoSSE62619.2024.00014. URL: https://ieeexplore.ieee.org/document/10608315.

[Cou+24] Mariana Coutinho u. a. „The Role of Generative AI in Software Development Productivity: A Pilot Case Study“. en. In: *Proceedings of the 1st ACM International Conference on AI-Powered Software*. Porto de Galinhas Brazil: ACM, Juli 2024, S. 131–138. ISBN: 9798400706851. DOI: 10.1145/3664646.3664773. URL: https://dl.acm.org/doi/10.1145/3664646.3664773.

[Cru22] Corbin Crutchley. *A Better Way To Code: Documentation Driven Development*. Jan. 2022. URL: https://playfulprogramming.com/posts/documentation-driven-development.

[DAD06] Sergio Cozzetti B. De Souza, Nicolas Anquetil und Káthia M. De Oliveira. „Which documentation for software maintenance?“ en. In: *Journal of the Brazilian Computer Society* 12.3 (Okt. 2006), S. 31–44. ISSN: 0104-6500, 1678-4804. DOI: 10.1007/BF03194494. URL: https://journal-bcs.springeropen.com/articles/10.1007/BF03194494.

[Dav+24] Nicole Davila u. a. „An Industry Case Study on Adoption of AI-based Programming Assistants“. In: *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ISSN: 2832-7659. IEEE, Apr. 2024, S. 92–102. DOI: 10.1145/3639477.3643648. URL: https://ieeexplore.ieee.org/document/10554736.

[Doh23] Thomas Dohmke. *The economic impact of the AI-powered developer lifecycle and lessons from GitHub Copilot*. en. Juni 2023. URL: https://github.blog/news-insights/research/the-economic-impact-of-the-ai-powered-developer-lifecycle-and-lessons-from-github-copilot/.

[DS24] Kyle Daigle und GitHub Staff. *Survey: The AI wave continues to grow on software development teams*. en. Aug. 2024. URL: https://github.blog/news-insights/research/survey-ai-wave-grows/.

[fre19] freeCodeCamp. *Sorting Algorithms Explained with Examples in JavaScript, Python, Java, and C++*. en. Dez. 2019. URL: https://www.freecodecamp.org/news/sorting-algorithms-explained-with-examples-in-python-java-and-c/.

[FRS16] Widia Resti Fitriani, Puji Rahayu und Dana Indra Sensuse. „Challenges in agile software development: A systematic literature review“. en. In: *2016 International Conference on Advanced Computer Science and Informati-*

on Systems (ICACSIS). Malang, Indonesia: IEEE, Okt. 2016, S. 155–164. ISBN: 978-1-5090-4629-4. DOI: 10.1109/ICACSIS.2016.7872736. URL: http://ieeexplore.ieee.org/document/7872736/.

[Git24] GitHub. *GitHub Copilot documentation*. en. Sep. 2024. URL: https://docs.github.com/en/copilot.

[Goo24] Google. *Google Python Style Guide*. en. Mai 2024. URL: https://android.googlesource.com/platform/external/google-styleguide/+/refs/tags/frc_350822020/pyguide.md.

[Gor04] Martin Gorrod. *Risk Management Systems*. en. London: Palgrave Macmillan UK, 2004. ISBN: 978-1-349-51263-8 978-0-230-51029-6. DOI: 10.1057/9780230510296. URL: http://link.springer.com/10.1057/9780230510296.

[GR24] Ya Gao und GitHub Customer Research. *Research: Quantifying GitHub Copilot's impact in the enterprise with Accenture*. en. Mai 2024. URL: https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/.

[GWW23] Zhijun Gao, Jiangying Wang und Meina Wang. „Mapping the Information Journey: Unveiling the Documentation Experience of Software Developers in China“. In: (2023). Version Number: 1. DOI: 10.48550/ARXIV.2312.02586. URL: https://arxiv.org/abs/2312.02586.

[Hee14] Lise Tordrup Heeager. „How Can Agile and Documentation-Driven Methods be Meshed in Practice?“ en. In: *Agile Processes in Software Engineering and Extreme Programming*. Hrsg. von Wil Van Der Aalst u. a. Bd. 179. Series Title: Lecture Notes in Business Information Processing. Cham: Springer International Publishing, 2014, S. 62–77. ISBN: 978-3-319-06861-9 978-3-319-06862-6. DOI: 10.1007/978-3-319-06862-6_5. URL: http://link.springer.com/10.1007/978-3-319-06862-6_5.

[HM24] Jon Hagar und Satoshi Masuda. „Prompt Engineering Impacts to Software Test Architectures for Beginner to Experts“. In: *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. ISSN: 2159-4848. Mai 2024, S. 116–121. DOI: 10.1109/ICSTW60967.2024.00034. URL: https://ieeexplore.ieee.org/document/10675959.

[HN09] Lise Tordrup Heeager und Peter Axel Nielsen. „Agile Software Development and its Compatibility with a Document-Driven Approach? A Case Study“. en. In: (2009). URL: http://aisel.aisnet.org/acis2009/84.

[Hou+24] Xinyi Hou u. a. „Large Language Models for Software Engineering: A Systematic Literature Review“. en. In: (Apr. 2024). arXiv:2308.10620 [cs]. URL: http://arxiv.org/abs/2308.10620.

[Kal22] Eirini Kalliamvakou. *Research: Quantifying GitHub Copilot's impact on developer productivity and happiness*. en. Sep. 2022. URL: https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/.

[Kal24] Eirini Kalliamvakou. *A developer's second brain: Reducing complexity through partnership with AI*. en-US. Jan. 2024. URL: https://github.blog/news-insights/research/a-developers-second-brain-reducing-complexity-through-partnership-with-ai/.

[Ken17] Seth Kenlon. *Why you should always do documentation before development*. en. Aug. 2017. URL: https://opensource.com/article/17/8/doc-driven-development.

[Knu99] Donald Ervin Knuth. *Literate Programming*. en. Juni 1999. URL: https://www-cs-faculty.stanford.edu/~knuth/lp.html.

[Li+24] Junjie Li u. a. „Fine Tuning Large Language Model for Secure Code Generation“. In: *2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering (Forge) Conference Acronym:* IEEE, Apr. 2024, S. 86–90. DOI: 10.1145/3650105.3652299. URL: https://ieeexplore.ieee.org/document/10599549.

[LSF03] T.C. Lethbridge, J. Singer und A. Forward. „How software engineers use documentation: the state of the practice“. en. In: *IEEE Software* 20.6 (Nov. 2003), S. 35–39. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1241364. URL: http://ieeexplore.ieee.org/document/1241364/.

[Luq+04] Luqi u. a. „Documentation driven development for complex real-time systems“. en. In: *IEEE Transactions on Software Engineering* 30.12 (Dez. 2004), S. 936–952. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.100. URL: http://ieeexplore.ieee.org/document/1377190/.

[Maj+24] Vahid Majdinasab u. a. „Assessing the Security of GitHub Copilot's Generated Code - A Targeted Replication Study“. In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. ISSN: 2640-7574. IEEE, März 2024, S. 435–444. DOI: 10.1109/SANER60148.2024.00051. URL: https://ieeexplore.ieee.org/document/10589764.

[Mar+24] Irina Mariasova u. a. *Developers save up to 8 hours per week with JetBrains AI Assistant*. en. Section: JetBrains AI. Apr. 2024. URL: https://blog.jetbrains.com/ai/2024/04/developers-save-up-to-8-hours-per-week-with-jetbrains-ai-assistant/.

[Mas+23] Antonio Mastropaolo u. a. „On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot“. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. ISSN: 1558-1225. IEEE, Mai 2023, S. 2149–2160. DOI: 10.1109/ICSE48619.2023.00181. URL: https://ieeexplore.ieee.org/document/10172792.

[Mir20] Stephen Miracle. *Secret to Good Programming? Documentation Driven Development*. en. Juli 2020. URL: https://www.linkedin.com/pulse/secret-good-programming-documentation-driven-stephen-miracle.

[Mus+22] Michel Muszynski u. a. „A Study on the Software Architecture Documentation Practices and Maturity in Open-Source Software Development“. en. In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. Honolulu, HI, USA: IEEE, März 2022, S. 47–57. ISBN: 978-1-66541-728-0. DOI: 10.1109/ICSA53651.2022.00013. URL: https://ieeexplore.ieee.org/document/9779701/.

[OBr+24] David OBrien u. a. „Are Prompt Engineering and TODO Comments Friends or Foes? An Evaluation on GitHub Copilot“. In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. ISSN: 1558-1225. Apr. 2024, S. 2707–2719. URL: https://ieeexplore.ieee.org/document/10549612.

[ODr22] Pippa O'Driscoll. *Using documentation-driven development for GOV.UK Sign In*. en. Mai 2022. URL: https://gds.blog.gov.uk/2022/05/09/using-documentation-driven-development-for-gov-uk-sign-in/.

[Pan+23] Weifeng Pan u. a. „Pride: Prioritizing Documentation Effort Based on a PageRank-Like Algorithm and Simple Filtering Rules“. en. In: *IEEE Transactions on Software Engineering* 49.3 (März 2023), S. 1118–1151. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2022.3171469. URL: https://ieeexplore.ieee.org/document/9765699/.

[Par+23] Daeseung Park u. a. „A Study on Performance Improvement of Prompt Engineering for Generative AI with a Large Language Model“. In: *Journal of Web Engineering* 22.8 (Nov. 2023). Conference Name: Journal of Web Engineering, S. 1187–1206. ISSN: 1544-5976. DOI: 10.13052/jwe1540-9589.2285. URL: https://ieeexplore.ieee.org/document/10452390.

[Par19] Jessica Parsons. *Lessons Learned in a Year of Docs-Driven Development*. Portland, Mai 2019. URL: https://www.youtube.com/watch?v=WDYQoZ-QDRM.

[PDS14] Reinhold Plosch, Andreas Dautovic und Matthias Saft. „The Value of Software Documentation Quality“. en. In: *2014 14th International Conference on Quality Software*. Allen, TX, USA: IEEE, Okt. 2014, S. 333–342. ISBN: 978-1-4799-7198-5 978-1-4799-7197-8. DOI: 10.1109/QSIC.2014.22. URL: http://ieeexplore.ieee.org/document/6958422/.

[Pre10] Tom Preston-Werner. *Readme Driven Development*. en. Aug. 2010. URL: https://tom.preston-werner.com/2010/08/23/readme-driven-development.

[Pro16] Daniele Procida. *Documentation-driven development*. en. 2016. DOI: 10.5446/21121. URL: https://av.tib.eu/media/21121.

[RBG22] Sawan Rai, Ramesh Chandra Belwal und Atul Gupta. „A Review on Source Code Documentation“. en. In: *ACM Transactions on Intelligent Systems and Technology* 13.5 (Okt. 2022), S. 1–44. ISSN: 2157-6904, 2157-6912. DOI: 10.1145/3519312. URL: https://dl.acm.org/doi/10.1145/3519312.

[RDC23] Alberto D. Rodriguez, Katherine R. Dearstyne und Jane Cleland-Huang. „Prompts Matter: Insights and Strategies for Prompt Engineering in Automated Software Traceability“. In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. ISSN: 2770-6834. IEEE, Sep. 2023, S. 455–464. DOI: 10.1109/REW57809.2023.00087. URL: https://ieeexplore.ieee.org/document/10260721.

[Ric14] Steve Richert. *On Documentation-Driven Development: Putting a bit more emphasis on your README*. en. Apr. 2014. URL: https://collectiveidea.com/blog/archives/2014/04/21/on-documentation-driven-development/.

[Rod23] Mario Rodriguez. *Research: Quantifying GitHub Copilot's impact on code quality*. en. Okt. 2023. URL: https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-code-quality/.

[Ros+13] Dominik Rost u. a. *Software Architecture Documentation for Developers: A survey*. en. Springer Berlin Heidelberg, Jan. 2013, S. 72–88. DOI: 10.1007/978-3-642-39031-9\{_}7. URL: https://doi.org/10.1007/978-3-642-39031-9_7.

[SA16] C. J. Satish und M. Anand. „Software Documentation Management Issues and Practices: A Survey". en. In: *Indian Journal of Science and Technology* 9.20 (Mai 2016). ISSN: 0974-5645, 0974-6846. DOI: 10.17485/ijst/2016/v9i20/86869. URL: https://indjst.org/articles/software-documentation-management-issues-and-practices-a-survey.

[Sam19] John Samuel. *Documentation-driven Development*. Sep. 2019. URL: https://johnsamuel.info/en/programming/documentation-driven-development.html.

[Sas+24] Yuya Sasaki u. a. „Systematic Literature Review of Prompt Engineering Patterns in Software Engineering". In: *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. ISSN: 2836-3795. Juli 2024, S. 670–675. DOI: 10.1109/COMPSAC61105.2024.00096. URL: https://ieeexplore.ieee.org/document/10633588.

[SJK16] Spencer Smith, Thulasi Jegatheesan und Diane Kelly. „Advantages, Disadvantages and Misunderstandings About Document Driven Design for Scientific Software". en. In: *2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE)*. IEEE, Nov. 2016, S. 41–48. DOI: 10.1109/SE-HPCCSE.2016.010. URL: https://ieeexplore.ieee.org/document/7839470.

[SMT23] John Ferguson Smart, Jan Molak und Daniel Terhorst-North. *BDD in action*. Second edition. Shelter Island, NY: Manning, 2023. 460 S. ISBN: 978-1-61729-753-3.

[Sta24] Stack Overflow. *2024 Stack Overflow Developer Survey*. en. Mai 2024. URL: https://survey.stackoverflow.co/2024/.

[Sup14] Zach Supalla. *Documentation-Driven Development (DDD)*. März 2014. URL: https://gist.github.com/zsup/9434452.

[SUW23] Lakmal Silva, Michael Unterkalmsteiner und Krzysztof Wnuk. „Towards identifying and minimizing customer-facing documentation debt“. en. In: *2023 ACM/IEEE International Conference on Technical Debt (TechDebt)*. Melbourne, Australia: IEEE, Mai 2023, S. 72–81. ISBN: 9798350311945. DOI: 10.1109/TechDebt59074.2023.00015. URL: https://ieeexplore.ieee.org/document/10207098/.

[URZ24] Ifiok Udoidiok, Hassan Reza und Jielun Zhang. „Exploring AI Integration in Software Development: Case Studies and Insights“. In: *NAECON 2024 - IEEE National Aerospace and Electronics Conference*. ISSN: 2379-2027. IEEE, Juli 2024, S. 375–380. DOI: 10.1109/NAECON61878.2024.10670347. URL: https://ieeexplore.ieee.org/document/10670347.

[Whi+23] Jules White u. a. „A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT“. In: (2023). Version Number: 1. DOI: 10.48550/ARXIV.2302.11382. URL: https://arxiv.org/abs/2302.11382.

[WZW21] Shengnan Wu, Yangfan Zhou und Xin Wang. „Exploring User Experience of Automatic Documentation Tools“. en. In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. Yokohama Japan: ACM, Mai 2021, S. 1–6. ISBN: 978-1-4503-8095-9. DOI: 10.1145/3411763.3451606. URL: https://dl.acm.org/doi/10.1145/3411763.3451606.

[Zha03] Lynn Zhang. „Documentation Driven Agile Development for Systems of Embedded Systems“. en. In: *Workshop on Software Engineering for Embedded Systems [SEES 2003]: From Requirements to Implementation*. Computer Programming and Software. DTIC, Sep. 2003, S. 13. URL: https://apps.dtic.mil/sti/citations/ADP015440.

[Zhi+15] Junji Zhi u. a. „Cost, benefits and quality of software development documentation: A systematic mapping“. en. In: *Journal of Systems and Software* 99 (Jan. 2015), S. 175–198. ISSN: 01641212. DOI: 10.1016/j.jss.2014.09.042. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121214002131.

A Anhang

A.1 Verwendete Hilfsmittel

In der Tabelle A.1 sind die im Rahmen der Bearbeitung des Themas der Bachelorarbeit verwendeten Werkzeuge und Hilfsmittel aufgelistet.

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

Tool	Verwendung
ChatGPT	Unterstützung bei redaktionellen Prozessen wie Grammatik- und Syntaxprüfungen, stilistischen Umformulierungen und punktuellen Übersetzungen von Fachbegriffen
Git	Verwaltung der Änderungen an der Bachelorarbeit
GitHub	Speichern der Bachelorarbeit
GitHub Copilot	Codegenerierung im Rahmen des Proof of Concepts
LaTeX	Erstellung dieses Dokuments
Numbers	Datenorganisation und -analyse
VS Code	Erstellen dieses Dokuments und des Proof of Concepts
Zotero	Verwaltung der Referenzen

A.2 CD Inhaltsverzeichnis

Auf der CD im Root-Verzeichnis befinden sich folgende Dateien:

- **`Bachelorarbeit.pdf`** - Digitale Version dieses Dokuments.
- **`NumSort.zip`** - Quellcode und Dokumentation des Proof of Concepts.

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original