Abschlussprüfung Sommer 2026

Software Developer (IHK)

Dokumentation zur Projektarbeit

EIS Schuh-Atelier

Entwicklung einer lokalen Desktop-Applikation zur Bestandsverwaltung (EIS Schuh-Atelier)

Abgabedatum: Saarbrücken, den 26.06.2026

Prüfungsbewerber:

Ekaterina Eisfeld

Ottweilerstraße 46

66113 Saarbrücken

Projektbetreuer:

Erwin Reichel

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Onlinequellenverzeichnis

1. **[Q1]** Python Software Foundation: *Python 3.14 Documentation.* URL: https://docs.python.org/3/ (abgerufen am: 10.05.2026)
2. **[Q2]** Riverbank Computing Limited: *PyQt6 Reference Guide.* URL: https://www.riverbankcomputing.com/static/Docs/PyQt6/ (abgerufen am: 09.05.2026)
3. **[Q3]** Hipp, D. R.: *SQLite Documentation.* URL: https://www.sqlite.org/docs.html (abgerufen am: 01.06.2026)
4. **[Q4]** van Rossum, G.; Warsaw, B.; Coghlan, A.: *PEP 8 – Style Guide for Python Code.* URL: https://peps.python.org/pep-0008/ (abgerufen am: 20.04.2026)
5. **[Q5]** PyInstaller Development Team: *PyInstaller Manual.* URL: https://pyinstaller.org/en/stable/ (abgerufen am: 19.05.2026)
6. **[Q6]** JGraph Ltd.: *draw.io – Diagramming Software.* URL: https://www.diagrams.net (abgerufen am: 07.06.2026)
7. **[Q7]** Wikipedia: *Wasserfallmodell.* URL: https://de.wikipedia.org/wiki/Wasserfallmodell (abgerufen am: 03.06.2026)
8. **[Q8]** Wikipedia: *Model View Controller.* URL: https://de.wikipedia.org/wiki/Model_View_Controller (abgerufen am: 06.05.2026)
9. **[Q9]** ISO 25000 Portal: *ISO/IEC 25010 – Product Quality Model.* URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (abgerufen am: 27.05.2026)
10. **[Q10]** Anthropic: *Claude – KI-Assistenzsystem.* URL: https://www.anthropic.com/claude (abgerufen am: 18.06.2026)
11. **[Q11]** Google: *Gemini – KI-Assistenzsystem.* URL: https://gemini.google.com (abgerufen am: 19.06.2026)
12. **[Q12]** OpenAI LLC: ChatGPT. URL: ChatGPT (abgerufen am: 21.06.2026)

Abkürzungs- und Glossarverzeichnis

Abkürzung	Ausgeschrieben	Bedeutung im Projektkontext
API	Application Programming Interface	Schnittstelle zwischen Softwarekomponenten und Bibliotheken
CRUD	Create, Read, Update, Delete	Grundlegende Datenbankoperationen der Anwendung
CSS	Cascading Style Sheets	Grundlage für die Gestaltung von QSS-Oberflächen
DB	Datenbank	Strukturierte Speicherung der Schuhdaten
ERM	Entity-Relationship-Modell	Modellierung der Datenstruktur der Anwendung
EXE	Executable File	Ausführbare Windows-Datei der Anwendung
GUI	Graphical User Interface	Grafische Benutzeroberfläche des EIS Schuh-Ateliers
IDE	Integrated Development Environment	Entwicklungsumgebung zur Softwareerstellung
IHK	Industrie- und Handelskammer	Prüfende Institution des Abschlussprojekts
ISO	International Organization for Standardization	Referenz für Qualitätsanforderungen
KI	Künstliche Intelligenz	Unterstützende Werkzeuge bei Recherche und Dokumentation
MVC	Model-View-Controller	Verwendetes Softwarearchitekturmuster
OOP	Objektorientierte Programmierung	Programmierparadigma von Python
PDF	Portable Document Format	Dokumentenformat für Projektunterlagen
PEP 8	Python Enhancement Proposal 8	Python-Richtlinie für sauberen Programmcode
PyQt6	Python Qt Version 6	Framework für die grafische Oberfläche
QComboBox	Qt ComboBox	Auswahlfeld für Eingaben in der Oberfläche
QDialog	Qt Dialog	Dialogfenster innerhalb der Anwendung
QPixmap	Qt Pixmap	Klasse zur Bilddarstellung
QSS	Qt Style Sheets	Gestaltungssprache für die Benutzeroberfläche
QTableWidget	Qt Table Widget	Tabellenkomponente für die Schuhübersicht
Regex	Regular Expression	Regulärer Ausdruck zur Datumsvalidierung
SQL	Structured Query Language	Sprache für Datenbankabfragen
SQLite	Structured Query Language Lite	Lokale Datenbank der Anwendung
TDD	Test Driven Development	Vorgehen bei automatisierten Tests
UI	User Interface	Benutzerschnittstelle der Anwendung
UML	Unified Modeling Language	Modellierungssprache für Diagramme
Unit-Test	Unit Test	Automatisierter Test einzelner Funktionen
URL	Uniform Resource Locator	Internetadresse einer Quelle
VS Code	Visual Studio Code	Eingesetzte Entwicklungsumgebung
XML	Extensible Markup Language	Strukturierte Auszeichnungssprache
Attribut	—	Eigenschaft eines Schuhdatensatzes
Build-Prozess	—	Erstellung der ausführbaren Anwendung
Controller	—	Steuerungsschicht innerhalb der MVC-Architektur
Datenmodell	—	Struktur der gespeicherten Schuhdaten
Datenpersistenz	—	Dauerhafte Speicherung von Daten
Datensatz	—	Einzelner Eintrag eines Schuhs
Deployment	—	Bereitstellung der fertigen Anwendung
Dropdown-Menü	—	Auswahlliste für Benutzereingaben
Framework	—	Softwaregrundlage zur Entwicklung der Anwendung
Freitextsuche	—	Suchfunktion für Schuhdaten
Geschäftslogik	—	Fachliche Verarbeitung von Eingaben und Daten
Logging	—	Protokollierung von Ereignissen und Fehlern
Mockup	—	Entwurf der Benutzeroberfläche
Primärschlüssel	—	Eindeutige Identifikation eines Datensatzes
Repository	—	Speicherort der Git-Versionierung

Validierung	—	Prüfung von Benutzereingaben
Versionsverwal-tung	—	Verwaltung von Codeänderungen mittels Git
Wireframe	—	Früher Entwurf der Benutzeroberfläche
Wasserfallmodell	—	Verwendetes Vorgehensmodell im Projekt

1 Einleitung

1.1 Projektumfeld

Das Projekt **„EIS Schuh-Atelier“** wird im Rahmen der Abschlussprüfung zum Software Developer (IHK) als eigenständige Projektarbeit durchgeführt. Da es sich um eine private Eigenentwicklung handelt, existiert kein klassischer externer Auftraggeber oder Ausbildungsbetrieb.

Die Entwicklerin übernimmt in diesem Projekt sowohl die Rolle der Auftraggeberin als auch die der späteren Hauptanwenderin. Die Anwendung wird ausschließlich im privaten Umfeld eingesetzt und dient der strukturierten, lokalen und datenschutzkonformen Verwaltung einer privaten Schuhsammlung.

Als Zielumgebung wurde bewusst ein lokales Einzelplatzsystem unter Microsoft Windows gewählt. Dadurch wird sichergestellt, dass die Anwendung ohne zusätzliche Infrastruktur, Internetverbindung oder externe Dienste betrieben werden kann und reale Einsatzbedingungen eines typischen Desktop-Systems widerspiegelt.

1.2 Projektziel

Ziel des Projekts ist die Entwicklung einer lokal ausführbaren Desktop-Applikation mit dem Namen „EIS Schuh-Atelier“ zur digitalen Inventarisierung einer privaten Schuhsammlung. Die Anwendung wird in `Python 3.14` umgesetzt und verwendet das Framework `PyQt6` zur Realisierung einer grafischen Benutzeroberfläche. Zur persistenten Speicherung der Daten kommt eine lokale `SQLite`-Datenbank zum Einsatz. Mit `PyInstaller` wird eine ausführbare `.exe`-Datei erzeugt.

Kernfunktionalitäten der Anwendung sind das Anlegen, Anzeigen, Bearbeiten und Löschen von Schuheinträgen (**CRUD**-Operationen). Jeder Eintrag beinhaltet Attribute wie das Pflichtfeld Marke sowie die optionalen Angaben zu Saison, Farbe, Preis, Kaufdatum und einem zugehörigen Bildpfad. Ergänzend werden eine Freitextsuche sowie kombinierbare Filterfunktionen für Marke, Farbe und Kaufjahr implementiert, die eine strukturierte und schnelle Übersicht über den gesamten Bestand ermöglichen.

1.3 Projektbegründung

Die Motivation für die Durchführung dieses Projekts ergibt sich aus dem stetig wachsenden Umfang der privaten Schuhsammlung und dem damit verbundenen Verlust der Übersicht. Bisher erfolgte die Verwaltung der vorhandenen Schuhe unstrukturiert, beispielsweise durch Erinnerungen oder unsortierte Fotografien.

Die Suche nach bestimmten Modellen – etwa nach Saison, Farbe oder Marke – war dadurch zeitaufwendig und fehleranfällig. Zusätzlich bestand die Gefahr von Fehl- oder Doppelkäufen, da kein zentraler Überblick über den vorhandenen Bestand und die bereits getätigten Anschaffungen existierte.

Durch die Applikation wird dieser Zustand durch eine digitale, filterbare Bestandsverwaltung ersetzt. Die Anwendung ermöglicht eine Zeitersparnis, erhöht die Übersichtlichkeit und reduziert das Risiko unnötiger Ausgaben. Zusätzlich ermöglicht die Anwendung eine transparente Übersicht über den Gesamtwert der Sammlung sowie über die finanziellen Ausgaben insgesamt und pro Jahr.

1.4 Projektschnittstellen

Die Desktop-Applikation „EIS Schuh-Atelier“ interagiert im Rahmen dieses Projekts mit verschiedenen technischen und organisatorischen Schnittstellen. Diese gewährleisten den reibungslosen Betrieb der Anwendung und ihre Nutzung im vorgesehenen Einsatzumfeld.

Benutzerschnittstelle (GUI):
Die primäre Schnittstelle zum Anwender ist die grafische Benutzeroberfläche, die mit dem Framework `PyQt6` realisiert wird. Über diese Oberfläche steuert der Benutzer sämtliche Funktionen der Anwendung, insbesondere das Anlegen, Bearbeiten, Löschen und Suchen von Schuheinträgen.

Datenbankschnittstelle:
Zur persistenten Speicherung der Anwendungsdaten kommuniziert die Geschäftslogik über `SQL`-Abfragen mit einer lokalen `SQLite-Datenbank`. In dieser Datei mit dem Pfad `database/schuhe.db` werden sowohl die Stammdaten der Schuhe als auch die Pfade zu den zugehörigen Bilddateien dauerhaft gesichert.

Betriebssystem-Schnittstelle:
Die Anwendung greift auf das lokale Dateisystem des Betriebssystems Microsoft Windows zu, um Bilddateien zu verwalten. Hierzu werden systemeigene Dateidialoge für die Auswahl und Speicherung der Fotos verwendet.

Organisatorische Schnittstelle:
Da es sich um ein privates Eigenprojekt handelt, erfolgte die Genehmigung des Projektrahmens sowie die Bewertung des Projektergebnisses durch den zuständigen Prüfungsausschuss der Industrie- und Handelskammer.

1.5 Projektabgrenzung

Um den vorgegebenen Projektzeitraum von 80 Stunden einzuhalten und eine zielgerichtete Umsetzung zu gewährleisten, wurde der Funktionsumfang der Anwendung bewusst begrenzt. Der Fokus lag auf der Umsetzung einer stabilen und benutzerfreundlichen Kernfunktionalität.

Nicht Bestandteil dieses Projekts sind insbesondere:

- eine Web-, Server- oder Cloud-Anbindung der Anwendung
- eine Mehrbenutzer- oder Rechteverwaltung
- eine automatisierte Bilderkennung oder KI-gestützte Kategorisierung
- die Anbindung an externe Bezahlsysteme oder Online-Shops

2 Projektplanung

2.1 Projektphasen

Die Durchführung des Projekts sollte innerhalb eines geplanten Gesamtzeitraums von **80 Stunden** erfolgen. Um eine strukturierte Umsetzung zu gewährleisten, wurde das Projekt von Anfang an in klar definierte, aufeinander aufbauende Phasen gegliedert (Vijayakumaran, 2024):

1. **Analysephase:** Betrachtung des Ist-Zustands sowie Definition der fachlichen und funktionalen Anforderungen.
2. **Entwurfsphase:** Erarbeitung der konzeptionellen Grundlagen (Architektur, Datenmodell, UI-Design).
3. **Implementierungsphase:** Technische Umsetzung der Anforderungen (Datenstrukturen, `PyQt6`-Oberfläche, Geschäftslogik).
4. **Abnahmephase:** Überprüfung der Funktionsfähigkeit und Abgleich mit den Anforderungen.
5. **Einführungs- und Dokumentationsphase:** Erstellung der Anwender- und Entwicklerdokumentation sowie finale Paketierung.

Projektphase	Geplante Zeit	Haupttätigkeiten
1. Analysephase	10 h	Ist-Analyse, Wirtschaftlichkeitsbetrachtung (Make-or-Buy), Erstellung des Lastenhefts
2. Entwurfsphase	15 h	Architekturdesign (MVC), Datenbankmodell (ERM), UI-Mockups, Erstellung des Datenverarbeitungskonzepts (anstatt Pflichtenhefts)
3.Implementierungsphase	30 h	Programmierung der `SQLite`-Anbindung, Geschäftslogik (CRUD), grafische `PyQt6`-Benutzeroberfläche
4. Abnahme- und Testphase	8 h	Funktionstests, automatisierte Verifikation der Datenbankschicht (Unit-Tests), Fehlerbehebung, Soll-/Ist-Vergleich
5. Dokumentationsphase	17 h	Erstellung der IHK-Projektdokumentation und der Kundendokumentation
Gesamt	**80 h**	Max. 80 Stunden gemäß IHK-Vorgabe (Implementierung max. 30 h)

Tabelle 1: Grobe Zeitplanung (gemäß genehmigtem Projektantrag)[1]

2.2 Abweichungen vom Projektantrag

Im Verlauf des Projekts ergaben sich zwei wesentliche Abweichungen hinsichtlich der technologischen Umgebung und des funktionalen Datenmodells im Vergleich zum ursprünglichen Projektantrag:

2.2.1 Aktualisierung der Entwicklungsumgebung (Python 3.14 statt 3.12)

- Geplante Python-Version: `Python 3.12`
- Tatsächlich verwendete Version: `Python 3.14`

Begründung: Während der direkt nach der Konzeption startenden Implementierungsphase wurde die bewusste Entscheidung getroffen, auf die aktuelle Python-Version 3.14 zu setzen. Dies geschah primär, um von den neuesten Performance-Optimierungen des Interpreters (insbesondere dem reduzierten Speicherbedarf bei lokalen Desktop-Anwendungen) sowie den stark verbesserten, feingranularen Fehlermeldungen (Enhanced Tracebacks) im Debugging-Prozess zu profitieren. Diese Features erhöhten die Entwicklungsgeschwindigkeit bei der Ausarbeitung der komplexen Live-Filterlogik signifikant und stärken nachhaltig die Zukunftsfähigkeit sowie Wartbarkeit der Softwarebasis. (Post, 2021)

2.2.2 Erweiterung des Datenmodells um das Attribut „Kaufdatum“

- Geplante Attribute (laut Konzept): Marke, Saison, Farbe, Preis, Bildpfad

[1] Eine detailliertere Zeitplanungstabelle ist im Anhang 11.1 zu sehen.

- Tatsächlich erweitertes Attribut: Kaufdatum (`kaufdatum` als TEXT)

Begründung: Während der detaillierten Anforderungsanalyse (Phase 1) und der anschließenden Entwurfsphase (Phase 2) stellte sich heraus, dass eine reine Filterung und Sortierung der Schuhsammlung nach „Saison“ für eine langfristige Bestandsverwaltung nicht präzise genug ist. Um dem Anwender eine fundierte Auswertung des zeitlichen Kaufverhaltens sowie eine exakte Filterung nach bestimmten Anschaffungsjahren zu ermöglichen, wurde das Datenmodell flexibel um das Feld `kaufdatum` erweitert.

Die Validierung erfolgt im Controller über reguläre Ausdrücke (`Regex`), um fehlerhafte Eingaben strukturell auszuschließen. Diese funktionale Erweiterung konnte nahtlos in die bestehende Architektur integriert werden und hatte keinerlei negativen Einfluss auf den geplanten Zeitrahmen. Abgesehen von dieser inhaltlichen Aufwertung wurden alle Meilensteine inhalts- und termingleich wie beantragt umgesetzt.

2.3 Ressourcenplanung

Für die Realisierung des Projektes wurden gezielt Hard- und Softwarekomponenten gewählt, die eine effiziente Entwicklung, eine strukturierte Architektur sowie eine einfache Bereitstellung der Anwendung gewährleisten.

2.3.1 Hardware und Entwicklungsumgebung

Die Entwicklung erfolgte auf einem lokalen Entwicklungssystem unter dem Betriebssystem **Microsoft Windows 11 Pro**. Als integrierte Entwicklungsumgebung (IDE) wurde **Visual Studio Code** eingesetzt.

2.3.2 Softwarekomponenten und Bibliotheken

Das Projekt basiert auf der Programmiersprache **Python (Version 3.14)**. Zur Umsetzung der Anforderungen wurden folgende Komponenten und Frameworks integriert:

- **GUI-Framework[2] (PyQt6):** Ein plattformübergreifendes Framework zur Erstellung der grafischen Benutzeroberfläche. (Starke, 2024) (Willman, 2022)
  - `QtWidgets`: Zur Implementierung von Fenstern, Formularen, Tabellen und Dialogen.
  - `QtCore`: Für die grundlegende Anwendungslogik, Signale/Slots sowie die Steuerung von Elementgrößen.
  - `QtGui`: Zur Einbindung von grafischen Ressourcen wie Bildern, Icons und Pixmaps.
- **Datenbanksystem:** Zum Einsatz kommt `SQLite3`, eine lokale, dateibasierte relationale Datenbank. Die Integration erfolgt über das Python-Standardmodul `SQLite3`. Die Datenhaltung ist strukturiert in der Datei `database/schuhe.db` hinterlegt. (Wolfinger, 2025)
- **Standardbibliotheken:**
  - `logging`: Zur strukturierten Protokollierung von Programmabläufen und Fehlern in `logs/app.log`.
  - `os` & `sys`: Zur Dateipfadverwaltung und Systemschnittstellenansprache (u. a. zur Erkennung der Laufzeitumgebung via `sys.frozen`).
  - `re`: Nutzung regulärer Ausdrücke zur Validierung von Datumsformaten.

[2] GIU-Framework - Bibliothek zur Erstellung grafischer Benutzeroberflächen (Siehe Abkürzungs- und Glossarverzeichnis)

- Für Diagrammerstellung wurde **draw.io** herangezogen.

2.3.3 Softwarearchitektur

Um eine strikte Trennung von Daten, Logik und Benutzeroberfläche zu gewährleisten, wurde die **Model-View-Controller-**Architektur **(MVC)** (Starke, 2024) angewandt:

- **Model (`model.py`):** Verantwortlich für die **Daten**bankschicht und die Datenmanipulation.
- **View (`view.py`):** Beinhaltet die Präsentationsschicht und die Definition der **GUI**-Elemente.
- **Controller (`controller.py`):** Bildet die **Logik-/Steuerung**sschicht, welche die Benutzerinteraktionen verarbeitet und die Kommunikation zwischen Model und View koordiniert.

2.3.4 Build-Prozess und Deployment

Für die Bereitstellung wurde `PyInstaller` eingesetzt, um das Projekt in eine eigenständige Windows-Executable (`EIS-Schuh-Atelier.exe`) zu kompilieren. Damit ist die Anwendung auf Zielsystemen ohne installierte Python-Umgebung lauffähig.

Im Vorfeld wurde auch das alternative Werkzeug `cx_Freeze` evaluiert. Die Entscheidung fiel jedoch aus folgenden Gründen auf `PyInstaller`:

- **Ein-Datei-Modus** (`--onefile`)**:** PyInstaller ermöglicht es, alle Abhängigkeiten, Bibliotheken und Ressourcen in eine einzige, kompakte `.exe`-Datei zu packen. `cx_Freeze` erzeugt standardmäßig eine größere Ordnerstruktur mit vielen Einzeldateien, was die Weitergabe an den Endnutzer unkomfortabler macht.
- **Automatische Erkennung von Abhängigkeiten:** `PyInstaller` verfügt über eine fortschrittlichere, automatische Analyse von importierten Bibliotheken (insbesondere bei komplexeren Packages), wodurch der Konfigurationsaufwand im Vergleich zu `cx_Freeze` erheblich sank.

2.3.5 Versionsverwaltung und Qualitätssicherung

Zur Sicherung der Entwicklungsstände wurde das Versionsverwaltungssystem **Git** eingesetzt. Dies ermöglichte die lokale Versionierung des Quellcodes und die Nachvollziehbarkeit aller Änderungen. (Vijayakumaran, 2024)

Der Einsatz von **Git** entspricht den Standards moderner Softwareentwicklung und trägt maßgeblich zur strukturierten und revisionssicheren Umsetzung des Projektes bei. (Post, 2021) (Siessegger, 2024)[3]

2.4 Entwicklungsprozess

Die Softwareentwicklung erfolgt nach einem linearen Vorgehensmodell in Anlehnung an das **Wasserfallmodell.**[4] Dieses Modell wurde gewählt, da die Anforderungen durch die private Nutzung klar definiert sind und die Phasen Analyse, Entwurf und Implementierung logisch aufeinander aufbauen. Dies gewährleistet eine strukturierte Dokumentation jedes Entwicklungsschrittes. (Vijayakumaran, 2024)

[3] Ausführliche Erklärung der Versionsverwaltung ist im Anhang 11.14 dargestellt.
[4] Die Darstellung des Wasserfallmodels befindet sich im Anhang 11.4.

3 Analysephase

3.1 Ist-Analyse

Die Verwaltung der privaten Schuhsammlung erfolgt derzeit weitgehend unstrukturiert. Ein zentrales System zur Erfassung und Verwaltung des Bestands existiert nicht. Informationen über vorhandene Modelle werden überwiegend anhand von Fotografien auf dem Smartphone oder durch persönliche Erinnerung vorgehalten. Die Suche nach bestimmten Schuhen, beispielsweise nach Marke, Farbe oder Saison, ist dadurch zeitaufwendig und erfordert häufig das manuelle Durchsuchen des gesamten Bestands.

Dieser Zustand führt im Alltag zu Ineffizienzen bei der Verwaltung der Sammlung. Da keine filterbare Übersicht vorhanden ist, können Neuanschaffungen nur schwer mit dem vorhandenen Bestand abgeglichen werden, wodurch das Risiko von Doppelkäufen steigt. Darüber hinaus fehlt eine transparente Übersicht über die bisher getätigten Ausgaben sowie den Gesamtwert der Sammlung, da Einkaufsdaten nicht systematisch dokumentiert werden. Eine lokale, datenschutzkonforme und durchsuchbare Inventarisierung des Bestands ist aktuell nicht vorhanden. Daraus ergibt sich der Bedarf an einer softwaregestützten Lösung zur strukturierten Erfassung, Verwaltung und Auswertung der Schuhsammlung.

3.2 Wirtschaftlichkeitsanalyse

Da es sich bei diesem Projekt um eine private Eigenentwicklung handelt, wird die Rentabilität nicht im klassischen betriebswirtschaftlichen Sinne (etwa durch Umsatzsteigerung) (Wöhe, Döring, & Brösel, 2023) gemessen. Der wirtschaftliche Nutzen für den Anwender definiert sich stattdessen durch zwei Faktoren: die erhebliche Zeitersparnis bei der Verwaltung der Sammlung und die direkte Vermeidung von finanziellen Verlusten durch Fehl- oder Doppelkäufe. Die getätigte „Investition“ besteht primär aus der aufgewendeten Entwicklungszeit, welche durch den langfristigen, täglichen Nutzen aufgewogen wird.

Es wurde die Entscheidung getroffen, auf die fiktive Kostenaufstellung zu verzichten.

3.2.1 Make or Buy-Entscheidung

Vor Beginn der Entwicklung wurde geprüft, ob bereits eine geeignete Standardsoftware zur Bestandsverwaltung von Schuhen auf dem Markt existiert, die den Anforderungen entspricht. (Tiemeyer, 2021) Zwar gibt es diverse Inventarisierungs- und Kleiderschrank-Apps (Buy-Lösungen), diese weisen jedoch für diesen spezifischen Anwendungsfall entscheidende Nachteile auf. Die meisten dieser Anwendungen basieren auf Cloud-Lösungen und erfordern eine Registrierung, was dem Wunsch nach strikter lokaler Datenhaltung und Privatsphäre widerspricht. Zudem sind viele Apps an monatliche Abonnement-Gebühren gebunden, mit unnötigen Funktionen (wie z. B. Social-Media-Anbindungen) überladen und lassen die gezielte Auswertung des Investitionswertes vermissen.

Eine Eigenentwicklung (Make) als lokale PyQt6-Desktop-Applikation bietet hingegen den wesentlichen Vorteil der exakten Anpassbarkeit. Die Software wird präzise auf die benötigten Kernfunktionen und Attribute (`Marke`, `Saison`, `Farbe`, `Preis`, `Bild`) zugeschnitten, ohne störenden Overhead. Da keine laufenden Lizenz- oder Serverkosten anfallen und die uneingeschränkte Datenhoheit durch die lokale `SQLite`-Datenbankdatei `database/schuhe.db` beim Anwender verbleibt, stellt die Eigenentwicklung

für den vorliegenden Anwendungsfall die wirtschaftlich sinnvollste und technisch geeignetste Lösung dar.

3.3 Nutzwertanalyse

Um den qualitativen Mehrwert des Projektes neben den rein finanziellen Aspekten zu verdeutlichen, wird eine Nutzwertanalyse durchgeführt. Hierbei wird der bisherige Ist-Zustand (Verwaltung über das eigene Gedächtnis und unsortierte Smartphone-Fotos) der geplanten Neuentwicklung (lokale PyQt6-Anwendung) gegenübergestellt.

Die Bewertungskriterien wurden anhand der persönlichen Anforderungen an die Bestandsverwaltung definiert und gewichtet.[5]

Bewertungs-kriterium	Gewichtung	Ist-Zustand (Gedächtnis)	Nutzwert (Ist)	Soll-Zustand (PyQt6-App)	Nutzwert (Soll)
Übersichtlichkeit	30 %	1	0,30	5	1,50
Zeitersparnis bei Suche	25 %	2	0,50	5	1,25
Vermeidung von Fehlkäufen	20 %	2	0,40	4	0,80
Kosten- und Wertermittlung	15 %	1	0,15	5	0,75
Datenschutz (lokale Datenhaltung)	10 %	5	0,50	5	0,50
Gesamt	**100 %**		**1,85**		**4,80**

Tabelle 2: Nutzwertanalyse

Ergebnis: Die Auswertung zeigt mit 4,80 zu 1,85 Punkten einen extremen qualitativen Sprung. Besonders in den hoch gewichteten Bereichen Übersichtlichkeit und Zeitersparnis bietet das „EIS Schuh-Atelier“ einen massiven Mehrwert, der die Durchführung des Projektes absolut rechtfertigt.

3.4 Anwendungsfälle / Anforderungsanalyse

Die geplante Desktop-Applikation „EIS Schuh-Atelier“ muss verschiedene Kern-Anwendungsfälle (Use Cases[6]) abdecken, die den alltäglichen Umgang mit der Schuhsammlung abbilden. Zu den primären Anwendungsfällen gehören:

- **Schuh erfassen (Create):** Der Anwender legt einen neuen Datensatz an (Foto, Marke, Saison, Farbe, Preis, Kaufdatum).
- **Bestand durchsuchen/filtern (Read):** Der Anwender sucht nach spezifischen Modellen anhand von ausgewählten Kriterien (z. B. "Sommer" und/oder "Rot").
- **Eintrag bearbeiten (Update):** Nachträgliche Korrektur von Attributen oder Austausch des Fotos.
- **Schuh entfernen (Delete):** Löschen eines Datensatzes, wenn ein Paar entsorgt oder verkauft wurde.
- **Gesamtwert berechnen:** Das System aggregiert automatisch die Einkaufspreise zur Wertermittlung gesamt und pro gewähltes Jahr.

[5] **Bewertungsskala:** 1 = sehr schlecht / nicht vorhanden · 5 = sehr gut / vollständig vorhanden
Nutzwert = Gewichtung × Bewertung | **Ergebnis: Soll-Zustand (4,80) übertrifft Ist-Zustand (1,85) um Faktor 2,6**

[6] Ein grafisches Use-Case-Diagramm zur Veranschaulichung der Anwendungsfälle befindet sich im Anhang 11.2 dieser Dokumentation

3.5 Qualitätsanforderungen

Gemäß der Norm für Softwarequalität (in Anlehnung an ISO/IEC 25010[7] / ehemals 9126-1) werden an das private Softwareprojekt folgende primäre Qualitätsanforderungen gestellt[8]:

- **Benutzbarkeit (Usability):** Da die Anwendung im privaten Alltag genutzt wird, muss die PyQt6-Oberfläche hochgradig intuitiv bedienbar sein. Fehleingaben müssen durch vordefinierte Dropdown-Menüs (statt Freitext) minimiert werden.
- **Zuverlässigkeit (Reliability):** Die lokale Datenspeicherung in der SQLite-Datenbank muss fehlerfrei und robust erfolgen, um den Verlust von mühsam erfassten Bestandsdaten zu verhindern.
- **Effizienz (Performance):** Suchanfragen und Filterungen bei einem wachsenden Bestand (z. B. > 100 Einträge) müssen verzögerungsfrei (in unter 1 Sekunde) auf dem Bildschirm aktualisiert werden.

4 Entwurfsphase

4.1 Zielplattform

Als Zielplattform für das „EIS Schuh-Atelier“ wurde ein lokales Desktop-System (Microsoft Windows) gewählt. Im Gegensatz zu cloudbasierten Web-Lösungen gewährleistet diese Entscheidung, dass alle Bestandsdaten und Preisinformationen ausschließlich auf der eigenen Hardware gespeichert werden. Dies verhindert die Weitergabe von Daten an Drittanbieter, stellt die persönliche Datenhoheit sicher und ermöglicht eine vollständige Unabhängigkeit von einer Internetverbindung.

Als Programmiersprache kommt `Python` zum Einsatz, da es sich durch eine hohe Entwicklungsgeschwindigkeit und ein exzellentes Ökosystem für Desktop-Anwendungen auszeichnet. Als Datenbanksystem fiel die Wahl auf `SQLite`. Da `SQLite` als dateibasierte, serverlose Datenbank direkt in Python integriert ist und keine separate Server-Installation erfordert, ist es eine hochperformante und ressourcenschonende Lösung für ein lokales Einzelplatzsystem.

4.2 Architekturdesign

Die Softwarearchitektur wird streng nach der etablierten Model-View-Controller-Architektur (**MVC**) aufgebaut.[9] (Starke, 2024) Diese Art der Architektur dient dazu, eine saubere technische Trennung zwischen Datenhaltung, grafischer Benutzeroberfläche und Geschäftslogik zu gewährleisten:

- **Model (Datenmodell) -** Dieser Bereich kapselt die SQLite-Datenbankverbindung. Hier werden ausschließlich die SQL-Statements für die CRUD-Operationen (Create, Read, Update, Delete) ausgeführt und Daten an den Controller zurückgegeben. (Post, 2021)

[7] (Post, 2021)
[8] (Post, 2021)
[9] Siehe Anhang 11.11 Klassendiagramm – MVC Architektur

- **View (Präsentation) -** Dies umfasst die gesamte grafische Darstellung, die mittels des Frameworks PyQt6 realisiert wird. Die View ist "dumm"[10] – sie zeigt lediglich Daten an und registriert Benutzerklicks, führt aber niemals selbst Datenbankschritte aus.
- **Controller (Steuerung) -** Er bildet das Bindeglied. Der Controller nimmt die Eingaben des Nutzers aus der View entgegen (z. B. den Klick auf den "Speichern"-Button), validiert die Eingabedaten und leitet die entsprechenden Schreibbefehle an das Model weiter.

Diese modulare Trennung stellt sicher, dass zukünftige Änderungen – beispielsweise ein komplett neues Design der Benutzeroberfläche – durchgeführt werden können, ohne die grundlegende Datenbanklogik antasten zu müssen.

Eigenschaft	Gewichtung	tkinter	wxPython	Kivy	PyQt6
Modernes Widget-Angebot	4	2	3	2	5
Dokumentation & Community	5	4	3	3	5
Styling / QSS-Unterstützung	4	1	2	2	5
Tabellenansicht (QTableWidget)	5	2	3	1	5
Signal-Slot-Konzept	3	2	2	2	5
PyInstaller-Kompatibilität	4	5	4	3	5
Lernkurve / Erlernbarkeit	3	5	3	3	3
Gesamt (Rohwert)	28	83	81	63	134
Nutzwert (normiert 1–5)		2,96	2,89	2,25	4,79

Tabelle 3: Entscheidungsmatrix — Auswahl des GUI-Frameworks[11]

Anhand der Entscheidungsmatrix in der Tabelle 3: Entscheidungsmatrix — Auswahl des GUI-Frameworks wurde PyQt6 als GUI-Framework ausgewählt. Es erzielte mit einem Nutzwert von 4,79 den höchsten Gesamtwert. Ausschlaggebend waren insbesondere das umfangreiche Widget-Angebot (`QTableWidget`, `QComboBox`, `QDialog`), die `QSS`-Unterstützung für das Dark-Wood-Design sowie die zuverlässige Kompatibilität mit PyInstaller zur Erzeugung der eigenständigen `.EXE`-Datei. `tkinter` bietet zwar die geringste Lernkurve, reicht aber für eine professionelle Tabellenansicht und individuelles Styling nicht aus. (Ernesti & Peter Kaiser, 2025)

4.3 Datenbankmodell

Aufgrund der klaren und überschaubaren Struktur der zu speichernden Informationen wurde ein schlankes relationales Datenbankmodell entworfen. (Taylor & Blum, 2025) Die lokale SQLite-Datenbank basiert auf einer zentralen Tabelle „`schuhe`“, welche in der Datei `database/schuhe.db` gespeichert wird.

Die Tabelle enthält folgende Attribute:

- `id: INTEGER PRIMARY KEY AUTOINCREMENT` – eindeutiger Primärschlüssel
- `marke: TEXT NOT NULL` – Markenname des Schuhs
- `saison: TEXT` – saisonale Zuordnung
- `farbe: TEXT` – farbliche Kategorisierung
- `preis: REAL` – Kaufpreis des Schuhs
- `kaufdatum: TEXT` – Kaufdatum in den Formaten `tt.mm.jjjj, mm.jjjj oder jjjj`
- `bildpfad: TEXT` – absoluter Dateipfad zum Schuhbild

[10] enthält keine Geschäftslogik
[11] **Bewertungsskala:** 1 = sehr schlecht / nicht vorhanden · 3 = ausreichend · 5 = sehr gut / vollständig

Das Attribut `marke` wurde als Pflichtfeld definiert. Für das Kaufdatum wurde der Datentyp TEXT gewählt, da SQLite keinen nativen DATE-Datentyp besitzt und gleichzeitig unterschiedliche Eingabeformate unterstützt werden müssen.

Bilder werden nicht direkt in der Datenbank gespeichert. Statt eines binären `BLOB`-Feldes wird lediglich der Dateipfad zur Bilddatei abgelegt. Dadurch bleibt die Datenbankdatei klein und performant, während die Bilder unabhängig von der Datenbank verwaltet werden können.

Das vollständige **Entity-Relationship-Modell** wurde im Anhang 11.5 visualisiert.

4.4 Entwurf der Benutzeroberfläche

Die grafische Benutzeroberfläche (GUI) wird als klassische Desktop-Anwendung konzipiert. Da das Programm im privaten Alltag genutzt wird, liegt der Fokus auf einer intuitiven Bedienung (Usability) und einer klaren visuellen Struktur. (Starke, 2024)

Das `Hauptfenster` wird in funktionale Bereiche unterteilt: Eine Filter- und Navigationsleiste zur schnellen Selektion, ein zentraler Inhaltsbereich (Galerieansicht der vorhandenen Schuhe) und ein Eingabe- bzw. Detailbereich zum Anlegen und Bearbeiten von Einträgen (inklusive Foto-Vorschau und Preis-Eingabe).

Um eine ansprechende und moderne Optik zu gewährleisten, werden zur Gestaltung sogenannte `Qt Style Sheets (QSS)` innerhalb von PyQt6 verwendet. Diese Technologie funktioniert analog zu CSS in der Webentwicklung und erlaubt die vollständige Individualisierung des Designs – von Hintergrundfarben über abgerundete Buttons bis hin zu interaktiven Hover-Effekten.

Die zugehörigen Oberflächenentwürfe – vom Wireframe über das UI-Mockup bis zum ersten Prototyp der geplanten Fenster – befinden sich im Anhang 11.7 dieser Dokumentation.

4.5 Geschäftslogik

Die Geschäftslogik (der Controller im MVC-Muster) regelt den Datenfluss zwischen der PyQt6-Benutzeroberfläche und der SQLite-Datenbank. Der wichtigste Prozess der Anwendung ist das fehlerfreie Erfassen eines neuen Schuhs.

Dieser Ablauf gestaltet sich wie folgt: Sobald der Anwender in der Oberfläche auf „Speichern“ klickt, greift die Validierungslogik. Es wird geprüft, ob das programmiertechnische Pflichtfeld Marke über das Dropdown-Menü ausgewählt oder händisch eingetippt wurde. Die optionalen Felder Saison und Farbe werden bei Nichtauswahl automatisch zu einem leeren String normalisiert, damit sie nicht blockierend wirken.

Zusätzlich wird das Preisfeld auf ein gültiges numerisches Format geprüft, wobei die Logik das deutsche Komma-Dezimaltrennzeichen automatisch in das Python-konforme Punkt-Format konvertiert. Falls ein Kaufdatum eingegeben wurde, erfolgt eine Validierung über reguläre Ausdrücke gegen drei flexible Formate (`jjjj`, `mm.jjjj`, `tt.mm.jjjj`). Ist ein Foto ausgewählt, wird der lokale Dateipfad extrahiert. Erst wenn alle Prüfungen erfolgreich verlaufen, formuliert die Logik das parametrisierte `INSERT`-Statement und übergibt die Daten sicher an das Model.

Die Integration in den privaten Arbeitsfluss ist nahtlos: Nach jedem Einkauf wird die Applikation gestartet, der neue Schuh binnen Sekunden erfasst und das System steht sofort für eine erneute Filter- oder Suchanfrage nach Freitext, Marke, Farbe oder Kaufjahr bereit.

Im Anhang 11.6 befindet sich ein **Aktivitätsdiagramm**, welches den genauen Programmablauf beim Speichern eines neuen Datensatzes modelliert.

4.6 Maßnahmen zur Qualitätssicherung

Um die im Lastenheft definierten Qualitätsanforderungen (insbesondere Zuverlässigkeit) zu erfüllen, wird eine zweistufige Qualitätssicherung durchgeführt:

1. **Automatisierte Unit-Tests:** Die Einfüge- und Lesefunktionen der Datenbankschicht werden unabhängig von der grafischen Oberfläche mithilfe der Python-Bibliothek `unittest` geprüft. Hierbei wird eine isolierte temporäre Datenbankdatei erzeugt, um zu verifizieren, dass Datensätze korrekt geschrieben und ausgelesen werden, ohne die Produktionsdaten zu gefährden. (Post, 2021)
2. **Manuelle System- und Anwendertests:** Die grafische Benutzeroberfläche wird durch gezielte Blackbox-Tests geprüft. Dabei werden gezielt Fehleingaben (z. B. Buchstaben im Preisfeld) provoziert, um sicherzustellen, dass die Applikation nicht abstürzt, sondern dem Nutzer eine verständliche Fehlermeldung ausgibt.

Ein **Auszug aus dem Testprotokoll** ist dem Anhang 11.10 beigefügt.

4.7 Lastenheft

Das im Vorfeld erstellte Lastenheft definiert die Anforderungen aus Sicht des Anwenders und legt detailliert fest, was die Software leisten soll. (Tiemeyer, 2021) Nachfolgend werden die fünf zentralen Muss-Kriterien (LH-01 bis LH-05) als **repräsentativer Auszug** aufgeführt, die erfolgreich im fertigen Programm umgesetzt wurden:

- **Muss-Kriterium 1:** Die Software muss als eigenständig lauffähige Applikation lokal unter Windows ausführbar sein, ohne dass eine Python-Installation oder eine Internetverbindung auf dem Zielrechner erforderlich ist.
- **Muss-Kriterium 2:** Es muss eine lokale relationale Datenbank (`SQLite`) zur persistenten Speicherung der Schuhdaten angebunden werden, um den Bestand dauerhaft zu sichern.
- **Muss-Kriterium 3:** Die Software muss die funktionale Möglichkeit bieten, zu jedem Datensatz einen lokalen Bildpfad zu verknüpfen (als optionales Merkmal für den Anwender). Falls ein Produktbild ausgewählt wurde, muss die Oberfläche dieses Foto sowohl als Vorschau im Erfassungsformular als auch als skalierte Miniaturansicht innerhalb der Bestandsliste visualisieren.
- **Muss-Kriterium 4:** Die grafische Oberfläche muss eine performante Filterung des Bestands erlauben. Dabei müssen mindestens vier Kriterien (Freitextsuche, Marke, Farbe und Jahr) gleichzeitig über eine logische UND-Verknüpfung gefiltert werden können.
- **Muss-Kriterium 5:** Die Anwendung muss den Gesamtwert („Schatzwert“) der Sammlung automatisch berechnen. Diese Aggregation muss dynamisch erfolgen

und sich bei aktiven Filtern in Echtzeit auf die aktuell sichtbaren Datensätze anpassen.

Das vollständige Lastenheft mit allen Kriterien (LH-01 bis LH-15) befindet sich im Anhang 11.3.

5 Implementierungsphase

5.1 Implementierung der Datenstruktur

Für die Datenhaltung wurde die in der Python-Standardbibliothek enthaltene `SQLite`-Schnittstelle `sqlite3` verwendet. Die gesamte Datenbanklogik wurde in der Klasse SchuhModell (`model.py`) gekapselt und entsprechend dem MVC-Architekturmuster von der Benutzeroberfläche getrennt.

Da ich bis zu diesem Projekt nur begrenzte praktische Erfahrung mit Datenbanken hatte, war diese Phase ein intensiver Lernprozess. Vor den ersten Codezeilen musste ich mich zunächst mit SQL-Abfragen und dem Modul `sqlite3` vertraut machen. Besonders zu Beginn war der Umgang mit Datenbankverbindungen ungewohnt: Ich hatte keine klare Vorstellung davon, wann eine Verbindung geöffnet und wann sie wieder geschlossen werden sollte, und ob es problematisch ist, dieselbe Verbindung in mehreren Methoden zu verwenden. Erst durch gezieltes Ausprobieren und die Lektüre der `sqlite3`-Dokumentation entwickelte ich ein Verständnis dafür, wie Verbindungsmanagement in einer Desktop-Anwendung sinnvoll umgesetzt wird.

Beim Start der Anwendung wird zunächst der Speicherort der Datenbank bestimmt. Da sich der Ablageort im Entwicklungsbetrieb und nach der Kompilierung mit PyInstaller unterscheidet, erfolgt eine Unterscheidung zwischen beiden Ausführungsmodi über `sys.frozen`. Dadurch wird sichergestellt, dass die Datenbankdatei dauerhaft neben der ausführbaren Anwendung gespeichert wird. Diese Unterscheidung war zunächst nicht auf meinem Radar – ich bemerkte das Problem erst, als ich die kompilierte `EXE-Datei` das erste Mal außerhalb meiner Entwicklungsumgebung testete und die Anwendung die Datenbank nicht finden konnte. Die Lösung über `sys.frozen` war eine direkte Konsequenz aus diesem konkreten Fehler.

Anschließend werden die benötigten Verzeichnisse `database/` und `logs/` automatisch mit `os.makedirs(..., exist_ok=True)` angelegt. Im Konstruktor der Klasse wird die Verbindung zur SQLite-Datenbank aufgebaut. Zusätzlich wird `row_factory = sqlite3.Row` gesetzt, wodurch Datenbankfelder über ihren Spaltennamen angesprochen werden können – beispielsweise `zeile["marke"]` statt über numerische Indizes. Diese Zeile wirkte beim ersten Lesen unscheinbar, erwies sich aber als einer der wertvollsten Handgriffe im gesamten Modell: Ohne sie war jeder Zugriff auf Datenbankfelder fehleranfällig, weil die Reihenfolge der Spalten entscheidend war.

Die Methode `_tabelle_erstellen()` erzeugt die Tabelle beim ersten Programmstart automatisch mittels `CREATE TABLE IF NOT EXISTS`. (Taylor & Blum, 2025) (Wolfinger, 2025) Bereits vorhandene Tabellen bleiben unverändert bestehen. Eine besondere Herausforderung trat auf, als ich während der Entwurfsphase erkannte, dass ein weiteres Attribut – das Kaufdatum – notwendig ist. Was zunächst wie eine kleine Ergänzung wirkte, stellte mich vor ein konkretes Problem: Ich hatte bereits Testdaten in der Datenbank, die ich nicht verlieren wollte. Ein einfaches Löschen und Neuerstellen der Tabelle war keine Option mehr. Die Lösung bestand darin, mittels `PRAGMA table_info(schuhe)` zu prüfen, welche Spalten bereits existieren, und

`kaufdatum` nur dann automatisch per `ALTER TABLE` nachzurüsten, wenn sie fehlt. (Wolfinger, 2025) Dadurch bleibt die Anwendung zu älteren Datenbankversionen kompatibel. Dieses Problem – und seine Lösung – hat mir mehr über Datenbankentwicklung beigebracht als jede theoretische Erklärung: Ich hatte erstmals erlebt, was es bedeutet, ein laufendes System weiterzuentwickeln, ohne vorhandene Daten zu gefährden. (Taylor & Blum, 2025)

Die CRUD-Operationen sowie weitere Datenbankzugriffe wurden in separaten Methoden implementiert: `schuh_hinzufuegen()`, `alle_schuhe_abrufen()`, `schuh_aendern()`, `schuh_loeschen()`, `alle_marken_abrufen()` und `verbindung_schliessen()`. Die Methode `alle_marken_abrufen()` liefert alle vorhandenen Markennamen ohne Duplikate in alphabetischer Reihenfolge und wird im Controller für die Filterfunktion verwendet.

Anfangs erschien mir die konsequente Auslagerung dieser Logik in eine eigene Klasse als unnötiger Mehraufwand. Im weiteren Projektverlauf zeigte sich jedoch, dass Änderungen an der Datenhaltung – wie die Einführung des Kaufdatums – deutlich einfacher umzusetzen waren, weil die Benutzeroberfläche davon vollständig unberührt blieb. Die Trennung, die mir zu Beginn abstrakt und überdimensioniert vorkam, rettete mir später mehrfach Zeit.

Sämtliche SQL-Anweisungen verwenden parametrisierte Platzhalter `(?)`. Dadurch werden Benutzereingaben niemals direkt in SQL-Anweisungen eingebettet und `SQL-Injection` wird verhindert. (Post, 2021) Das **Datenverarbeitungskonzept** befindet sich im Anhang 11.15.

5.2 Implementierung der Benutzeroberfläche

Die grafische Oberfläche wurde vollständig in der Klasse `Hauptfenster` implementiert, die von `QMainWindow` erbt (`view.py`). Die Entwicklung der Benutzeroberfläche mit PyQt6 war für mich der anspruchsvollste Teil des gesamten Projekts – und gleichzeitig der lehrreichste. Vor diesem Projekt hatte ich noch keinerlei praktische Erfahrung mit einem GUI-Framework. Das Konzept, dass eine Oberfläche nicht sequenziell von oben nach unten aufgebaut wird, sondern aus ineinander verschachtelten Containern besteht, die sich gegenseitig beeinflussen, war für mich zunächst schwer greifbar.

Der Aufbau der Oberfläche wurde mit `QVBoxLayout` und `QHBoxLayout` als ineinander verschachtelten Layout-Containern organisiert, sodass sich das Fenster bei Größenänderungen responsiv anpasst. Zusammengehörige Elemente wurden mit `QGroupBox`-Containern gruppiert – das Eingabeformular, die Filterleiste und die Tabellenansicht bilden drei logisch getrennte Bereiche. Insbesondere das Zusammenspiel dieser Layout-Container bereitete mir anfangs erhebliche Schwierigkeiten: Mehrfach kam es vor, dass Elemente nicht an der erwarteten Position dargestellt wurden oder sich bei Größenänderungen des Fensters unerwartet verlagerten. Ich baute dasselbe Layout mehrfach neu auf – nicht weil der Code falsch war, sondern weil ich noch kein mentales Modell davon hatte, wie sich verschachtelte Layouts gegenseitig beeinflussen. Der Durchbruch kam, als ich begann, die Layout-Hierarchie vor dem Coden auf Papier zu skizzieren. Ab diesem Punkt entwickelte ich ein deutlich besseres Verständnis dafür, warum ein Element an einer bestimmten Stelle landet.

Eine weitere Herausforderung war das Konzept von Signalen und Slots. In der Dokumentation klang es einfach, aber in der Praxis war es anfangs schwer nachzuvollziehen, warum ein Signal manchmal ausgelöst wurde, bevor die Oberfläche vollständig

aufgebaut war, und welche Konsequenzen das für die Reihenfolge der Initialisierung hatte. Ich musste lernen, die Verbindung zwischen Signalen und Controller-Methoden erst herzustellen, nachdem alle Widgets vollständig initialisiert waren – ein Detail, das mir mehrere schwer erklärbare Fehler einbrachte, bevor ich die Ursache verstand.

Das visuelle Design wurde bewusst von der Struktur getrennt und zentral als `Qt Style Sheet (QSS)` in `view.py` definiert. QSS funktioniert analog zu CSS im Web: Selektoren für Widget-Typen (`QMainWindow, QGroupBox, QPushButton`) legen Farben, Schriften, Ränder und Hover-Effekte fest. Das dunkle Holzbraun `(#2C1503)` als Hintergrundfarbe und Gold `(#C9A84C, #D4AF37)` als Akzentfarbe werden konsistent durch die gesamte Anwendung genutzt. Über `QPushButton:hover` erhalten Schaltflächen eine visuelle Rückmeldung beim Überfahren mit dem Mauszeiger, ohne dass dafür Ereignis-Handler im Python-Code erforderlich sind. (Ernesti & Peter Kaiser, 2025)

Zu Beginn wirkte die Anwendung zwar funktional, aber optisch wie ein technischer Rohbau. Die Standarddarstellung von PyQt6 – graue Hintergründe, eckige Schaltflächen, kein visuelles System – entsprach in keiner Weise dem, was ich mir für eine Anwendung namens „Schuh-Atelier" vorgestellt hatte. Der Einstieg in QSS war dabei nicht trivial: Einige Selektoren verhielten sich anders als erwartet, bestimmte Widgets ignorierten Stile, die auf ihre Eltern-Container angewendet wurden, und die Wechselwirkungen zwischen verschiedenen Selektoren waren nicht immer vorhersehbar. Ich arbeitete das Stylesheet iterativ heraus – Eigenschaft für Eigenschaft – und testete jede Änderung direkt. Der Moment, in dem die Anwendung erstmals das gewünschte Erscheinungsbild hatte und tatsächlich wie ein Produkt und nicht wie ein Prototyp wirkte, war für mich einer der befriedigendsten Punkte des gesamten Projekts. (Fitzpatrick, 2025)

Für die Detailansicht und Bearbeitung eines Eintrags wurde ein separates Detailfenster als `QDialog` implementiert, das modal über dem Hauptfenster erscheint und die Daten des angeklickten Schuhs vorausgefüllt präsentiert. Durch die Arbeit mit PyQt6 habe ich gelernt, wie umfangreich moderne GUI-Frameworks aufgebaut sind – und dass ein solches Framework nicht in wenigen Tagen vollständig durchdrungen wird. Ich habe gelernt, gezielt mit der Dokumentation zu arbeiten und Lösungen iterativ zu entwickeln, anstatt eine vollständige Antwort vorauszusetzen, bevor der erste Code geschrieben ist. (Ernesti & Peter Kaiser, 2025)

Die konzeptionellen Oberflächenentwürfe und UI-Mockups befinden sich in Anhang 11.7; die Screenshots der fertigen Anwendung sind in Anhang 11.8 dokumentiert.

5.3 Implementierung der Geschäftslogik

Die Steuerungsschicht liegt vollständig in `SchuhController (controller.py)`, der als Bindeglied zwischen `SchuhModell` und `Hauptfenster` dient. Beide Schichten kennen sich nicht direkt – dies entspricht dem MVC-Prinzip. Zu Beginn dieser Phase fiel es mir schwer, die Verantwortlichkeiten klar zu trennen. Ich stellte mir wiederholt die Frage, ob eine bestimmte Logik in den Controller oder ins Model gehört. Konkret: Sollte die Validierung einer Benutzereingabe im Controller stattfinden oder in der Methode, die den Datensatz speichert? Ich entschied mich dafür, die Validierung vollständig im Controller zu halten, damit das Model ausschließlich für den Datenbankzugriff zuständig bleibt und unabhängig testbar ist. Diese Entscheidung zahlte sich später in der Testphase aus.

Der Einstiegspunkt ist `_signale_verbinden()`. (Robbins, 2025) Dort wird jedes interaktive Widget der View mit einer Controller-Methode verknüpft: Ein Klick auf den Speichern-Button löst `schuh_speichern()` aus; jede Änderung in einem der vier Filterfelder (`textChanged` bzw. `currentTextChanged`) ruft `_filter_anwenden()` auf. Eine Schwierigkeit, die ich in dieser Phase unterschätzte, war die Reihenfolge: Wenn Signale zu früh verbunden werden – bevor alle Widgets vollständig initialisiert sind – lösen sie unbeabsichtigt aus und führen zu Fehlern, die im Code selbst nicht sichtbar sind. Ich lernte dadurch, dass die Reihenfolge der Initialisierung in einer ereignisgesteuerten Anwendung keine Nebensächlichkeit ist. (Kofler, 2024)

Die Filterung erfolgt im Arbeitsspeicher, nicht per wiederholter Datenbankabfrage: `_filter_anwenden()` durchläuft alle Tabellenzeilen und setzt für jede `setRowHidden()`. Die vier Kriterien (`Freitext, Marke, Farbe, Kaufjahr`) sind `UND`-verknüpft – eine Zeile bleibt sichtbar, wenn alle aktiven Filter erfüllt sind. Meine erste Idee war, bei jeder Filteränderung eine neue Datenbankabfrage mit `WHERE`-Bedingungen abzusetzen. Diese Lösung schien zunächst naheliegend, weil Datenbanken für genau solche Aufgaben gebaut sind. Im weiteren Verlauf erkannte ich jedoch, dass bei kleinen Datenmengen der Overhead des wiederholten Datenbankzugriffs größer ist als der Aufwand der In-Memory-Filterung – und dass `setRowHidden()` mir gleichzeitig erlaubt, den Schätzwert direkt aus den sichtbaren Zeilen zu berechnen, ohne eine zweite Abfrage zu benötigen. Diese Erkenntnis war für mich ein gutes Beispiel dafür, dass die offensichtlichste technische Lösung nicht immer die praktischste ist.

Der Schätzwert wird am Ende derselben Methode neu berechnet, indem ausschließlich die Preiszellen sichtbarer Zeilen summiert werden. Ist kein Filter aktiv, wird der vorberechnete Gesamtwert aller Datensätze angezeigt; bei aktivem Filter werden gefilterter Wert und Gesamtwert nebeneinander dargestellt.

Die Eingaben werden in `schuh_speichern()` in mehreren Schritten validiert: Zunächst werden Platzhalter-Texte zu leeren Strings normalisiert, anschließend wird geprüft, ob eine Marke eingetragen wurde – sie ist das einzige Pflichtfeld. Das Kaufdatum wird per regulärem Ausdruck validiert, der drei Formate akzeptiert (`tt.mm.jjjj`, `mm.jjjj`, `jjjj`). Der Preis wird tolerant umgewandelt: Ein Komma als Dezimaltrennzeichen wird `vor float()` durch einen Punkt ersetzt, damit beide Schreibweisen akzeptiert werden. Viele dieser Sonderfälle entdeckte ich erst beim Testen – ich gab Daten so ein, wie ich es intuitiv tat, und stellte fest, dass mein erster Validierungsentwurf diese Fälle nicht abdeckte. Jede solche Situation führte zu einer gezielten Erweiterung der Logik. Am Ende war mir klar: Eine Validierung ist nie beim ersten Entwurf vollständig – sie wächst mit dem Verständnis dafür, wie eine Anwendung tatsächlich benutzt wird. Mehr zur Implementierung im Anhang 11.16 Screenshots für Implementierung der Geschäftslogik.

5.4 Fehlerbehandlung und Debugging

Kritische Operationen – insbesondere alle Datenbankzugriffe und das Laden von Bilddateien – wurden konsequent mit try-except-Blöcken umgeben. Im Model werden SQLite-Fehler bereits im Konstruktor abgefangen; schlägt die Verbindung fehl, wird der Fehler geloggt und die Ausnahme weitergegeben, sodass die Anwendung kontrolliert abbricht statt in einem inkonsistenten Zustand zu verbleiben.

Da Fehlerbehandlung und strukturiertes Debugging in meiner Weiterbildung zu den zuletzt behandelten Themen gehörten, war dieser Aspekt der Implementierung der für mich konzeptionell neueste. Im Unterricht hatten wir die drei Fehlertypen – Syntaxfehler, Laufzeitfehler und Logikfehler – kennengelernt und Debugging-Werkzeuge wie Breakpoints, Step Into, Step Over, Call Stack und Watches in WebStorm praktisch erprobt. Diese Grundlage half mir, Fehler systematischer einzuordnen: Statt bei jedem unerwarteten Verhalten ratlos zu sein, fragte ich mich zunächst, um welche Fehlerart es sich handelt – das allein grenzte die Suche deutlich ein.

Die Übertragung auf Python und PyQt6 war jedoch nicht trivial. Anders als im Unterricht, wo wir Breakpoints direkt in WebStorm gesetzt hatten, verhielt sich PyQt6 beim Debuggen oft eigenwillig: Laufzeitfehler innerhalb der Ereignisschleife von Qt wurden manchmal still verschluckt, statt eine lesbare Fehlermeldung zu erzeugen. Besonders beim Zusammenspiel der drei Schichten war nicht sofort erkennbar, ob die Ursache in der Datenbankabfrage, im Controller oder in der Darstellung durch die View lag. Der naheliegende Ansatz – einzelne `print()`-Ausgaben an verdächtigen Stellen – funktionierte für isolierte Probleme, stieß aber bei komplexeren Abläufen schnell an seine Grenzen. Ein weiteres Problem zeigte sich in der kompilierten EXE-Datei, in der `print()`-Ausgaben gar nicht mehr sichtbar sind – ein Fehler, der nur dort auftrat, war auf diesem Weg nicht reproduzierbar.

Diese Erfahrung war der konkrete Anlass, mich mit dem Python-Modul `logging` auseinanderzusetzen. Jede Schicht (`model.py, view.py, controller.py`) initialisiert einen eigenen Logger mit `logging.getLogger(name)`. Fehler werden mit Schweregrad und Zeitstempel in die Log-Datei geschrieben – zum Beispiel: `logger.error("Fehler beim Laden der Daten: %s", e)`. Die Log-Datei übernahm damit die Rolle, die im Unterricht der Call Stack in WebStorm hatte: Sie macht den Programmablauf nachvollziehbar – auch dort, wo kein Debugger verfügbar ist. Anhand der Modulnamen und Einträge ließ sich der Ablauf auch in der kompilierten Anwendung vollständig rekonstruieren. (Post, 2021)

Auch die Frage, wo eine Ausnahme abgefangen werden soll, war zunächst nicht intuitiv. Im Unterricht hatten wir uns hauptsächlich mit isolierten Funktionen beschäftigt; in einer mehrschichtigen Anwendung ist die Entscheidung komplexer: Ein zu früh abgefangener Fehler verliert oft den Kontext, der für eine sinnvolle Reaktion notwendig ist. Ich entschied mich dafür, Ausnahmen bewusst weiterzugeben, wenn die aktuelle Schicht keine Möglichkeit hat, angemessen zu reagieren – und sie erst dort zu behandeln, wo eine konkrete Antwort möglich ist.

Technische Ausnahmen werden für die Anwenderin in verständliche Dialoge übersetzt: `QMessageBox.warning()` zeigt bei Datenbankfehlern, ungültigen Eingaben oder fehlenden Pflichtfeldern eine konkrete Fehlermeldung an, anstatt den Fehler stillschweigend zu ignorieren oder die Anwendung abstürzen zu lassen. Die Entscheidung, welche Fehler dem Nutzer angezeigt und welche nur protokolliert werden, erforderte Überlegung: Technische Details sind für eine Anwenderin ohne IT-Hintergrund nicht hilfreich – eine Fehlermeldung muss beschreiben, was zu tun ist, nicht was intern schiefgelaufen ist.

Rückblickend war die Auseinandersetzung mit Fehlerbehandlung und Debugging die Phase, in der ich am meisten über den tatsächlichen Ablauf der eigenen Anwendung

gelernt habe. Die Konzepte aus dem Unterricht – Fehlertypen, Debugging-Werkzeuge, systematische Analyse – bildeten dabei eine wichtige Grundlage. Ihre eigentliche Bedeutung erschloss sich mir jedoch erst durch die praktische Anwendung in einem realen Projekt.

5.5 Implementierung der automatisierten Unit-Tests (TDD)

Zur Absicherung der Datenbankschicht wurde der Entwicklungsprozess für die Methode `schuh_hinzufuegen()` nach dem Prinzip des Test-Driven Development (TDD) gestaltet. TDD folgt einem zyklischen Ablauf aus drei Phasen — Red, Green, Refactor — und stellt sicher, dass jede neue Funktionalität nur so weit implementiert wird, wie es ein zuvor definierter Test verlangt.

In meiner Weiterbildung hatten wir Unit-Tests anhand von Jest in JavaScript kennengelernt und dabei die Arrange-Act-Assert-Struktur als Grundprinzip erarbeitet: Zunächst wird der Testzustand vorbereitet (Arrange), dann die zu testende Operation ausgeführt (Act), und schließlich das Ergebnis geprüft (Assert). Für dieses Projekt entschied ich mich bewusst für das Python-Standardframework `unittest` statt für PyTest. Da die gesamte Anwendung ohne externe Abhängigkeiten auskommt und als EXE ohne Installation lauffähig ist, passte `unittest` als Teil der Standardbibliothek konsequent zu diesem Ansatz. Die AAA-Struktur aus dem Unterricht ließ sich dabei direkt auf den TDD-Zyklus übertragen: setUp() entspricht dem Arrange, der Methodenaufruf dem Act, die assert-Anweisung dem Assert.

Um Seiteneffekte auf die Produktionsdatenbank auszuschließen, wird in `setUp()` vor jedem Testlauf ein temporäres Verzeichnis mit einer isolierten Testdatenbank angelegt. Nach Abschluss jedes Tests gibt `tearDown()` alle Ressourcen frei und löscht das Verzeichnis vollständig. Dadurch ist jeder Test in sich geschlossen und in beliebiger Reihenfolge wiederholbar. Dieses Muster kannte ich aus dem Unterricht als Grundprinzip guter Testisolation — es in der eigenen Anwendung umzusetzen, machte den Unterschied zwischen Theorie und Praxis spürbar.

Der vollständige Testcode wurde unter Einsatz von KI-Assistenzsystemen generiert; die fachliche Kontrolle, Einbettung in das Projekt sowie die Verifikation der Testergebnisse erfolgten eigenständig. Diese Vorgehensweise entspricht dem in Anhang 11.17 KI-Offenlegung dokumentierten KI-Einsatz.

Der TDD-Zyklus gliedert sich in drei dokumentierte Phasen:

Phase 1 - Red: Zunächst wurde ein Test formuliert, bevor die zu testende Methode existierte. `test_phase1_red_schuh_hinzufuegen_existiert_noch_nicht()` prüfte mittels `hasattr(), ob schuh_hinzufuegen()` im Objekt vorhanden ist. Beim ersten Ausführen schlug dieser Test erwartungsgemäß fehl — der Test war rot. Dieser Moment war für mich ungewohnt: Im Unterricht hatten wir Tests immer für bereits vorhandenen Code geschrieben. Einen Test absichtlich scheitern zu lassen fühlte sich zunächst falsch an. Erst durch den TDD-Ansatz wurde mir klar, warum dieser Fehlschlag notwendig ist — er belegt, dass der Test tatsächlich etwas Neues überprüft und nicht zufällig grün ist.

Phase 2 - Green: Im nächsten Schritt wurde eine minimale Implementierung in `model.py` ergänzt — ausschließlich so viel Code, wie nötig war, um `test_phase2_green_schuh_wird_gespeichert()` zu bestehen. Dieser Test fügte einen Datensatz ein und prüfte, ob anschließend genau ein Eintrag in der Datenbank vorhanden war. Die Disziplin, wirklich nur das Minimum zu implementieren, war

schwieriger als erwartet: Der Impuls, gleich alle geplanten Features zu schreiben, ist stark. TDD erzwingt eine Fokussierung, die ungewohnt ist, aber verhindert, dass Code entsteht, der nicht durch Tests abgedeckt ist.

Phase 3 - Refactor: Nach dem erfolgreichen Testdurchlauf wurde der Code bereinigt: `row_factory = sqlite3.Row` wurde eingeführt, die Datenmigration für `kaufdatum` ergänzt und die Pfadlogik für den EXE-Betrieb hinzugefügt. Der Test `test_phase3_refactor_alle_felder_korrekt_gespeichert()` validierte anschließend alle fünf Datenbankfelder (`Marke, Saison, Farbe, Preis, Kaufdatum`) einzeln. Alle Tests blieben grün — das Verhalten blieb unverändert, der Code wurde jedoch nachhaltig verbessert. Diese Phase machte den praktischen Wert von Tests am deutlichsten sichtbar: Ich konnte den Code verändern und war sofort sicher, dass nichts Bestehendes kaputt gegangen war. Das vollständige Testprotokoll mit Konsolenausgabe befindet sich in Anhang 11.10.

6 Abnahmephase

In der Abnahmephase wurde die fertige Applikation „EIS Schuh-Atelier“ einer abschließenden Prüfung unterzogen, um sicherzustellen, dass alle im Lastenheft definierten Anforderungen erfüllt sind. Da es sich um eine Eigenentwicklung handelt, wurde die Abnahme formal durch den Entwickler in der Rolle des Auftraggebers durchgeführt.

Im Rahmen der Qualitätssicherung wurden zunächst alle automatisierten Unit-Tests für die Datenbank-Logik erfolgreich durchlaufen. Danach folgte ein umfassender Systemtest der PyQt6-Oberfläche. Dabei wurde verifiziert, dass die Filterfunktion über alle vier Kriterien `(Freitext, Marke, Farbe, Kaufjahr)` hinweg korrekte Datensätze liefert und der Gesamt-Investitionswert („`Schatzwert`“) bei jeder Änderung der Preisdaten präzise neu berechnet wird. Da alle Muss-Kriterien fehlerfrei umgesetzt wurden, konnte die Software offiziell abgenommen werden. Ein entsprechendes Prüf- und Abnahmeprotokoll ist im Anhang 11.10 dieser Dokumentation hinterlegt.

7 Einführungsphase

Da die Anwendung für den privaten, lokalen Gebrauch konzipiert wurde, gestaltet sich die Einführungsphase hochgradig effizient und nutzerfreundlich. Um die Abhängigkeit von einer lokal installierten Python-Entwicklungsumgebung auf dem Zielsystem zu eliminieren, wurde der vollständige Quellcode inklusive aller benötigten Bibliotheken (wie `PyQt6`) mittels des Tools `PyInstaller` in eine eigenständig lauffähige, ausführbare Datei (`.exe`) kompiliert.

Die Software kann somit als klassisches Desktop-Programm direkt unter Windows gestartet werden. Bei der erstmaligen Ausführung der Anwendung prüft die Logik das Vorhandensein der Datenbank. Ist diese nicht existent, initialisiert das Programm automatisch die benötigte SQLite-Datenbankdatei `schuhe.db` im Unterordner `database` des aktuellen Verzeichnisses. Es sind somit keinerlei manuelle Konfigurationen, Server-Setups oder administrative Datenbank-Installationen durch den Nutzer erforderlich (Zero-Configuration-Ansatz).

Um den reibungslosen Umgang mit der neuen Software zu gewährleisten, wurde begleitend eine kompakte Benutzerdokumentation erstellt. Diese **Bedienungsanleitung** visualisiert Schritt für Schritt die wichtigsten Kernfunktionen (Anlegen, Filtern, Löschen)

und liegt dem Projekt als Anlage 11.12 als **Auszug** bei. Damit ist die Software offiziell in den produktiven Alltag überführt.

8 Dokumentation

Um die langfristige Nutzbarkeit und Wartbarkeit des „EIS Schuh-Ateliers“ zu gewährleisten, wurde die Dokumentation strikt nach Zielgruppen getrennt. Es wird zwischen der Anwenderdokumentation für den Endnutzer und der technischen Dokumentation für Entwickler unterschieden.

8.1 Benutzerdokumentation

Für den alltäglichen Einsatz der Software wurde ein kompaktes Benutzerhandbuch erstellt. Da sich dieses an den privaten Endanwender richtet, wurde bei der Verfassung bewusst auf komplexe IT-Fachbegriffe (wie SQL-Statements, MVC-Muster oder Datentypen) verzichtet.

Die Anleitung ist stark visuell geprägt und führt den Nutzer anhand von kommentierten Screenshots Schritt für Schritt durch die Kernprozesse der Anwendung. Dazu gehören das Starten der ausführbaren Datei, das fehlerfreie Anlegen eines neuen Schuh-Eintrags über die Dropdown-Menüs sowie die Nutzung der Filter- und Löschfunktionen.

Ein Ausschnitt aus der erstellten Benutzerdokumentation befindet sich im Anhang 11.12.

8.2 Entwicklerdokumentation

Die technische Dokumentation richtet sich an IT-Fachkräfte und dient der zukünftigen Wartung oder Erweiterung der Software. Sie wurde direkt im Python-Quellcode (Inline-Dokumentation) realisiert.

Die gesamte Codebasis ist konsequent nach den Richtlinien von `PEP 8` strukturiert, um eine standardisierte und fehlerfreie Lesbarkeit zu garantieren. Darüber hinaus wurden sämtliche Module, Klassen und Methoden lückenlos mit detaillierten Docstrings gemäß `PEP 257` versehen. Diese beschreiben exakt die erwarteten Übergabeparameter, die Rückgabewerte und die Funktionsweise der Methoden, insbesondere im Bereich der Datenbankanbindung und der Signal-Slot-Verbindungen von PyQt6.

Ein repräsentativer Auszug des kommentierten Quellcodes ist dem Anhang 11.9 beigefügt.

9 Fazit

9.1 Soll-/Ist-Vergleich

Ein zentraler Bestandteil des Projektabschlusses ist der Abgleich der erzielten Ergebnisse mit den zu Beginn definierten Anforderungen. Das im Lastenheft festgelegte Hauptziel – die Entwicklung einer vollständig funktionsfähigen, lokal ausführbaren Desktop-Applikation zur Verwaltung einer privaten Schuhsammlung – wurde vollständig erreicht. Sämtliche Muss-Kriterien der funktionalen Anforderungen konnten erfolgreich umgesetzt werden.

Auch die zeitlichen Vorgaben wurden eingehalten. Die geplante Gesamtdauer von 80 Stunden sowie die maximale Vorgabe von 30 Stunden für die Implementierungsphase wurden nicht überschritten. Während der Projektdurchführung zeigte sich, dass einzelne Arbeitspakete weniger Zeit in Anspruch nahmen als ursprünglich geplant. So konnte die Make-or-Buy-Analyse aufgrund der geringen Anzahl vergleichbarer Anwendungen schneller abgeschlossen werden. Ebenso reduzierte sich der Aufwand für die Erstellung des Lastenhefts sowie für die Fehlerbehebung, da während der Testphase keine kritischen Fehler identifiziert wurden. Die dadurch frei gewordenen Stunden wurden gezielt für die Weiterentwicklung der Benutzeroberfläche und die Verfeinerung der UI-Mockups eingesetzt. Dadurch konnte die Benutzerfreundlichkeit und das visuelle Erscheinungsbild der Anwendung verbessert werden, ohne den vorgesehenen Projektzeitrahmen zu überschreiten. (Siehe Anhang 11.13, Tabelle 9: Soll-/Ist-Vergleich - Zeitplanung)

Das fertige Produkt übertrifft die ursprüngliche Planung in mehreren Punkten. Während zunächst eine funktionale Verwaltungsanwendung im Vordergrund stand, entstand im Verlauf des Projekts eine vollwertige Desktop-Applikation mit professionellem Erscheinungsbild, intuitiver Bedienung und einem konsistenten Gestaltungskonzept.

Ein besonderer Schwerpunkt lag auf der Entwicklung der grafischen Benutzeroberfläche. Für die Gestaltung des finalen Dark-Wood-Designs mit Gold-Akzenten wurden KI-gestützte Werkzeuge unterstützend eingesetzt. Diese halfen dabei, ein stimmiges Farbkonzept zu entwickeln und das `Qt Style Sheet (QSS)` schrittweise zu optimieren. Die fachliche Bewertung, Auswahl und Umsetzung der Gestaltungselemente erfolgten dabei eigenständig. Durch diese Unterstützung konnte innerhalb des vorgegebenen Zeitrahmens eine Oberfläche realisiert werden, die die ursprünglichen Designvorstellungen deutlich übertraf. (Siehe Anhang 11.13, Tabelle 8: Soll-/Ist-Vergleich - Anforderungen)

Zusammenfassend wurden alle funktionalen und technischen Anforderungen erfolgreich umgesetzt. Der geplante Zeitrahmen konnte eingehalten werden und die entwickelte Anwendung bietet einen erkennbaren Mehrwert gegenüber dem ursprünglichen Ist-Zustand. Das Projektziel wurde vollständig erreicht; das Endergebnis übertraf in mehreren Bereichen die anfänglichen Erwartungen.

9.2 Lessons Learned

Die Umsetzung des Projekts „EIS Schuh-Atelier" lieferte wertvolle fachliche und organisatorische Erkenntnisse, die für die weitere Tätigkeit als Software Developer von nachhaltigem Nutzen sind:

- **Zeitplanung und Projektfokus:** Die strikte Vorgabe, die reine Implementierungsphase auf maximal 30 Stunden zu begrenzen, erforderte ein hohes Maß an Disziplin. Die größte Lektion war hierbei die frühzeitige Reduzierung der Komplexität. Der Wechsel von der bereits vor der Antragsstellung verworfenen Web-Idee hin zu einer fokussierten PyQt6-Desktop-Anwendung war eine entscheidende und lehrreiche Maßnahme, um Ressourcen zu schonen und das Projekt im vorgegebenen Zeitrahmen erfolgreich abzuschließen.
- **Vorteile der eingesetzten Architektur:** Der konsequente Einsatz des Model-View-Controller-Musters (MVC) hat sich als enormer Vorteil erwiesen. Durch die

strikte Trennung von Benutzeroberfläche und Datenbanklogik war die Fehlersuche (`Debugging`) wesentlich einfacher und gezielter möglich, als wenn der Code vermischt gewesen wäre.

- **Umgang mit Frameworks:** Die Arbeit mit PyQt6 zeigte, wie effizient sich moderne Benutzeroberflächen gestalten lassen. Insbesondere das Signal-Slot-Konzept zum Abfangen von Nutzerinteraktionen erwies sich nach einer kurzen Einarbeitungszeit als extrem logisch und stabil. Zudem erwies sich der gezielte und transparente Einsatz von KI-Assistenzsystemen zur Code-Optimierung als wertvolle moderne Arbeitsmethode.

9.3 Ausblick

Obwohl die Anwendung in der aktuellen Version 1.0 alle im Lastenheft definierten Anforderungen vollständig erfüllt und produktiv nutzbar ist, bietet die modulare Code-Basis viel Raum für zukünftige Erweiterungen.

Für kommende Versionen des „EIS Schuh-Ateliers“ sind folgende Weiterentwicklungen denkbar:

- **Daten-Export:** Die Implementierung einer Funktion zum Export der gesamten Bestandsdaten als CSV-Datei, um diese bei Bedarf in Tabellenkalkulationsprogrammen weiterverarbeiten zu können.
- **Grafische Auswertungen:** Zur besseren Visualisierung des Investitionswertes könnten grafische Statistiken (z. B. Tortendiagramme zur Verteilung der Schuhmarken) über die Bibliothek `matplotlib` direkt in die PyQt6-Oberfläche integriert werden.
- **Erweitertes Löschkonzept (Soft-Delete & Archiv):** Das aktuelle, unwiderrufliche Löschen per `SQL-DELETE`-Befehl soll in einer zukünftigen Version durch ein „Soft-Delete“-Verfahren ersetzt werden. Dabei werden gelöschte Schuhe über ein Archiv-Flag in der Datenbank aufbewahrt, um dort als Historie abrufbar zu bleiben und eine Berechnung der durchschnittlichen Nutzungsdauer sowie des finanziellen Gesamtverlusts zu ermöglichen.
- **Automatisches Backup:** Ein Hintergrund-Skript, das die lokale SQLite-Datenbankdatei (`schuhe.db`) in regelmäßigen Abständen als Sicherheitskopie speichert.
- **Mobile Portierung:** Eine langfristige Umsetzung der Kernlogik als Smartphone-App für Android wird als nächste Entwicklungsstufe angestrebt.
- **Zentrale Bildverwaltung:** Anstatt absolute Dateipfade zu speichern, soll die Anwendung in einer zukünftigen Version Bilddateien beim Hinzufügen automatisch in einen zentralen Unterordner (z. B. `images/`) innerhalb des Anwendungsverzeichnisses kopieren. Gespeichert wird anschließend ausschließlich der relative Dateiname. Dadurch bleibt die Anwendung auch nach einem Verschieben des Programmordners oder einer Neuinstallation vollständig funktionsfähig, da keine Abhängigkeiten zu externen Pfaden entstehen.

10 Literaturverzeichnis

Ernesti, J., & Peter Kaiser. (2025). *Python 3 - Das Umfassende Handbuch.* Bonn: Rheinwerk Verlag GmbH. Abgerufen am Juni 2026

Fitzpatrick, M. (2025). *Create GUI Applications with Python & Qt6 (PyQt6 Edition).* London: Independently Published.

Kofler, M. (2024). *Python - Der Grundkurs.* Bonn: Rheinwerk Verlag GmbH.

Post, U. (2021). *Besser coden : Best Practices für Clean Code.* Bonn: Rheinwerk Computing.

Robbins, P. (2025). *Python Programmierung für Einsteiger 2025.* Polland, Polland: Amazon. Abgerufen am Juni 2026

Siessegger, N. (2024). *Git – kurz & gut.* Heidelberg: O'Reilly. Abgerufen am 2026

Starke, G. (2024). *Effektive Softwarearchitekturen - Ein praktischer Leitfaden.* München: Carl Hanser Verlag.

Taylor, A. G., & Blum, R. (2025). *SQL für dummies - Alles-in-einem-Band.* Weinheim: Wiley-VCH GmbH. Abgerufen am Juni 2026

Tiemeyer, E. (2021). *Handbuch IT-System- und Plattformmanagement.* München: Carl Hanser Verlag. Abgerufen am Juni 2026

Vijayakumaran, S. (2024). *DevOps - Wie IT-Projekte mit einem modernen Toolset und der richtigen Kultur gelingen.* Bonn: Rheinwerk Verlag GmbH. Abgerufen am Juni 2026

Willman, J. M. (2022). *Beginning PyQt - A Hands-on Approach to GUI Programming with PyQT6.* Sunnyvale: APress Media.

Wöhe, G., Döring, U., & Brösel, G. (2023). *Einführung in die Allgemeine Betriebswirtschaftslehre.* München: Vahlen. Abgerufen am 2026

Wolfinger, P. (2025). *SQLite 3 - Essentials.* Oettingen i. Bay.: PWP-Verlag.com.

Eidesstattliche Erklärung

Ich, Ekaterina Eisfeld, versichere hiermit, dass ich die vorliegende Dokumentation zur Projektarbeit

Entwicklung einer lokalen Desktop-Applikation zur Bestandsverwaltung (EIS Schuh-Atelier)

selbständig verfasst habe. Alle verwendeten Quellen und Hilfsmittel — einschließlich Internetrecherchen — sind vollständig angegeben. Wörtliche und sinngemäße Zitate wurden als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und nicht veröffentlicht.

> **Erklärung zur Nutzung von KI-Assistenzsystemen:**
> Im Rahmen dieser Projektarbeit habe ich KI-Assistenzsysteme (Claude, Gemini, Copilot, ChatGPT) als methodische Hilfsmittel eingesetzt. Die Nutzung umfasste die sprachliche Optimierung von Dokumentationstexten, die Unterstützung bei der Code-Analyse, Code-Optimierung, Code-Vorschlägen sowie die Erstellung von Diagramm-, Tabellen- und Abbildungsentwürfen. Sämtliche fachlichen Entscheidungen, die Programmierung, die Systemarchitektur sowie alle inhaltlichen Aussagen dieser Dokumentation wurden von mir eigenständig erarbeitet, kritisch geprüft und verantwortet. Die KI-Nutzung ist im Anhang (KI-Offenlegung)[12] vollständig dokumentiert.

Saarbrücken, den 26.06.2026

Ekaterina Eisfeld

[12] Siehe ergänzend Anhang 11.17 KI-Offenlegung.

11 Anhang

11.1 Detaillierte Zeitplanung

Nr.	Phase / Teilaufgabe	Beschreibung	Std.
1.1	**Ist-Analyse**	Analyse des aktuellen Zustands (keine digitale Schuhverwaltung vorhanden). Beschreibung der Probleme und Defizite.	2 h
1.2	**Make-or-Buy-Entscheidung**	Vergleich: Eigenentwicklung vs. bestehende Software-Lösungen. Begründete Entscheidung für Eigenentwicklung, Nutzwertanalyse..	2 h
1.3	**Lastenheft**	Erstellung des Lastenhefts mit Muss-/Soll-/Kann-Anforderungen.	6 h
2.1	**Architekturdesign**	Festlegung des MVC-Musters. Zuordnung der Klassen zu den drei Schichten.	5 h
2.2	**Datenbankmodell**	Entwurf des ER-Diagramms und des Tabellenmodells (Tabelle schuhe).	2 h
2.3	**UI-Mockups**	Erstellung von Papier-Skizzen und digitalen Mockups der Benutzeroberfläche.	5 h
2.4	**Entwicklungsumgebung einrichten**	Installation Python 3.14, PyQt6, SQLite, VS Code, Git-Repository initialisieren.	3 h
3.1	**Model implementieren**	`SchuhModell`-Klasse: SQLite-Verbindung, Tabellenerstellung, Migration, CRUD-Methoden.	6 h
3.2	**View implementieren**	`Hauptfenster` und DetailFenster mit PyQt6: Layout, Widgets, QSS-Styling (Dark Wood).	10 h
3.3	**Controller implementieren**	`SchuhController`: Signal-Slot-Verbindungen, Validierung, Filterlogik, Schatzwert-Berechnung.	8 h
3.4	**Integration und Debugging**	Zusammenführung aller Komponenten, Fehlersuche, Logging einrichten.	4 h
3.5	**PyInstaller-Build**	EXE-Erzeugung mit PyInstaller, Testen der kompilierten Anwendung.	2 h
4.1	**Funktionstest**	Manuelles Testen aller Use Cases (Anlegen, Bearbeiten, Löschen, Filtern, Suchen).	3 h
4.2	**Unit-Tests (TDD)**	Erstellung, Ausführung und Code-Optimierung der automatisierten Unit-Tests (`test_model.py`) zur Absicherung der CRUD-Logik.	3 h
4.3	**Fehlerbehebung**	Behebung identifizierter Fehler, Nachbesserungen.	2 h
5.1	**IHK-Dokumentation**	Ausformulierung aller Kapitel (1–9) der Projektdokumentation.	10 h
5.2	**Kundendokumentation**	Erstellung des Benutzerhandbuchs (Installation, Bedienung, Screenshots).	3 h
5.3	**Anhänge und Diagramme**	Erstellung aller Diagramme (UML, ER, Use-Case, Aktivität) und Anhänge.	4 h
	Gesamt	Gemäß IHK-Vorgabe: max. 80 Stunden gesamt, max. 30 h Implementierung	**80**

Tabelle 4: Detaillierte Soll-Zeitplanung[13]

Hinweis: Die tatsächlich benötigten Zeiten werden im Soll-/Ist-Vergleich (Kapitel 9.1 / Anhang 11.13) dokumentiert. Die Implementierungsphase hält mit 30 Stunden die IHK-Obergrenze ein.

11.2 Use-Case-Diagramm

Zeigt alle Anwendungsfälle der Applikation aus Sicht des Benutzers. Grüne Ellipsen markieren eingeschlossene Teilfunktionen (`include`), rot kennzeichnet den Bestätigungsdialog beim Löschen.

[13] Zurück zum Kapitel 2.1 Projektphasen

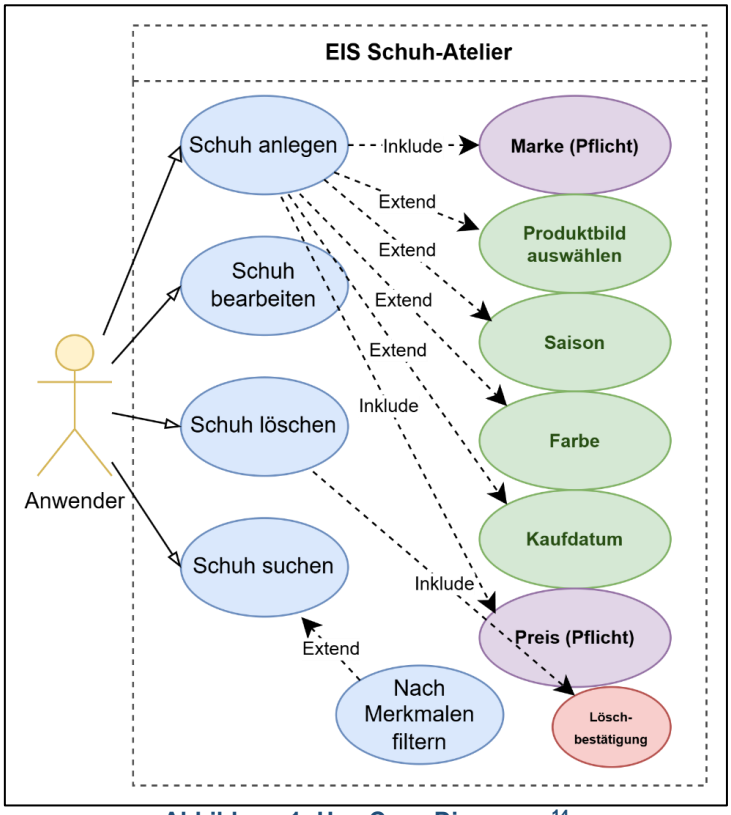


Abbildung 1: Use-Case-Diagramm[14]

11.3 Lastenheft (Auszug)

Anforderungen

Die Anforderungen sind in drei Prioritätsstufen gegliedert. Kapitel 4.7 der Projektdokumentation zitiert die Muss-Kriterien LH-01 bis LH-05 wortgleich als Kernbestandteil dieses Lastenhefts.

Nr.	Typ	Anforderung	Priorität
LH-01	Muss	Die Software muss als eigenständig lauffähige Applikation lokal unter Windows ausführbar sein, ohne dass eine Python-Installation oder eine Internetverbindung auf dem Zielrechner erforderlich ist.	Hoch
LH-02	Muss	Es muss eine lokale relationale Datenbank (SQLite) zur persistenten Speicherung der Schuhdaten angebunden werden, um den Bestand dauerhaft zu sichern.	Hoch
LH-03	Muss	Zu jedem Datensatz kann ein lokaler Bildpfad verknüpft werden. Falls ein Bild ausgewählt wurde, muss die Oberfläche dieses sowohl als Vorschau im Erfassungsformular als auch als Miniaturansicht innerhalb der Bestandsliste visualisieren.	Hoch
LH-04	Muss	Die grafische Oberfläche muss eine performante Filterung des Bestands erlauben.	Hoch

[14] Zurück zum Kapitel 3.4 Anwendungsfälle / Anforderungsanalyse

		Dabei müssen mindestens vier Kriterien (Freitextsuche, Marke, Farbe und Jahr) gleichzeitig über eine logische UND-Verknüpfung gefiltert werden können.	
LH-05	Muss	Die Anwendung muss den Gesamtwert („Schatzwert“) der Sammlung automatisch berechnen. Diese Aggregation muss dynamisch erfolgen und sich bei aktiven Filtern in Echtzeit auf die aktuell sichtbaren Datensätze anpassen.	Hoch
LH-06	Muss	Das vollständige Erfassen eines Schuheintrags muss möglich sein (Marke, Saison, Farbe, Preis, Kaufdatum). Die Marke ist Pflichtfeld.	Hoch
LH-07	Muss	Bestehende Einträge müssen über einen Bearbeitungsdialog geändert werden können.	Hoch
LH-08	Muss	Einträge müssen nach Bestätigung eines Sicherheitsdialogs gelöscht werden können.	Hoch
LH-09	Muss	Eingaben (Pflichtfeld Marke, Preisformat, Datumsformat) müssen vor dem Speichern validiert werden. Ungültige Eingaben werden mit einer verständlichen Fehlermeldung abgewiesen.	Hoch
LH-10	Soll	Eigenständig eingegebene Marken sollen nach dem Neustart erneut in der Auswahlliste erscheinen (dynamische Marken-Dropdown).	Mittel
LH-11	Soll	Die Tabelle soll nach allen Spalten sortierbar sein (Klick auf Spaltenüberschrift).	Mittel
LH-12	Soll	Alle Ereignisse und Fehler sollen in einer Logdatei (logs/app.log) protokolliert werden.	Mittel
LH-13	Kann	Der gefilterte Bestand soll als CSV-Datei exportiert werden können.	Niedrig
LH-14	Kann	Grafische Auswertungen (Schuhe pro Marke, Saison) als Diagramm.	Niedrig
LH-15	Kann	Mehrsprachigkeit: Oberfläche wahlweise auf Deutsch oder Englisch.	Niedrig

Tabelle 5: Muss-Soll-Kann-Anforderungen[15]

Die Muss-Kriterien LH-01 bis LH-05 werden in Kapitel 4.7 **Lastenheft** der Projektdokumentation wortgleich zitiert und als erfolgreich umgesetzt ausgewiesen.

[15] Legende: Muss = zwingend erforderlich | Soll = stark empfohlen | Kann = optional

11.4 Wasserfallmodell

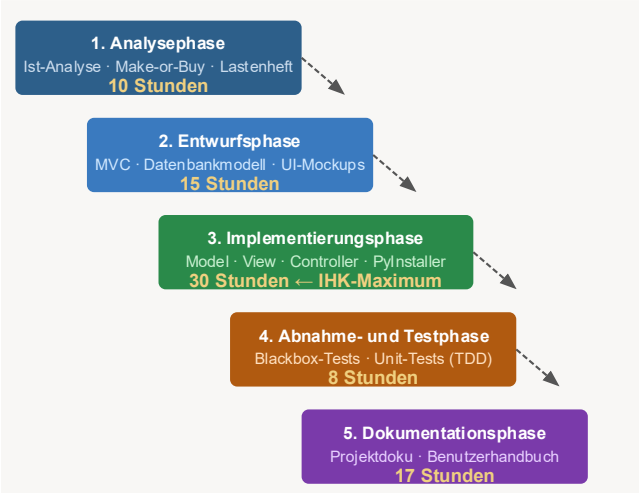


Abbildung 2: Wasserfallmodell — EIS Schuh-Atelier (Phasen und Stundenverteilung)

11.5 Entity-Relationship-Modell (ERM)

Das ERM besteht aus einer einzigen Entität `schuhe`, da die Anwendung bewusst als schlankes Einzelplatzsystem ohne relationale Verknüpfungen konzipiert wurde.

`id`	INTEGER	`PRIMARY KEY AUTOINCREMENT`	Eindeutiger Primärschlüssel, wird von SQLite automatisch vergeben
`marke`	TEXT	`NOT NULL`	Zwingendes Pflichtfeld für den Markennamen des Schuhs
`saison`	TEXT	nullable	Optionale Angabe zur saisonalen Zuordnung
`farbe`	TEXT	nullable	Optionale Angabe zur farblichen Kategorisierung
`preis`	REAL	nullable	Numerisches Feld zur Berechnung des Gesamtinvestitionswerts („Schatzwert“)
`kaufdatum`	TEXT	nullable	Optionales Kaufdatum in den flexiblen Formaten `tt.mm.jjjj`, `mm.jjjj` oder `jjjj`; Typ `TEXT`, da SQLite keinen nativen DATE-Typ besitzt
`bildpfad`	TEXT	nullable	Absoluter Dateipfad zur lokalen Bilddatei

Abbildung 3: Inhalte Datenbankmodell[16]

Die Spalte `kaufdatum` wurde nachträglich zum Datenbankschema hinzugefügt. Um Kompatibilität mit bereits bestehenden Installationen zu gewährleisten, prüft die Anwendung beim Start über `PRAGMA`[17] `table_info(schuhe)`, ob die Spalte bereits vorhanden ist. Fehlt sie, wird sie automatisch per `ALTER TABLE` ergänzt. So funktionieren ältere Datenbanken ohne manuelle Eingriffe weiterhin.

[16] Zurück zum Kapitel 4.3 Datenbankmodell.
[17] Zurück zum Kapitel 5.1 Implementierung der Datenstruktur.

11.6 Aktivitätendiagramm (UML)

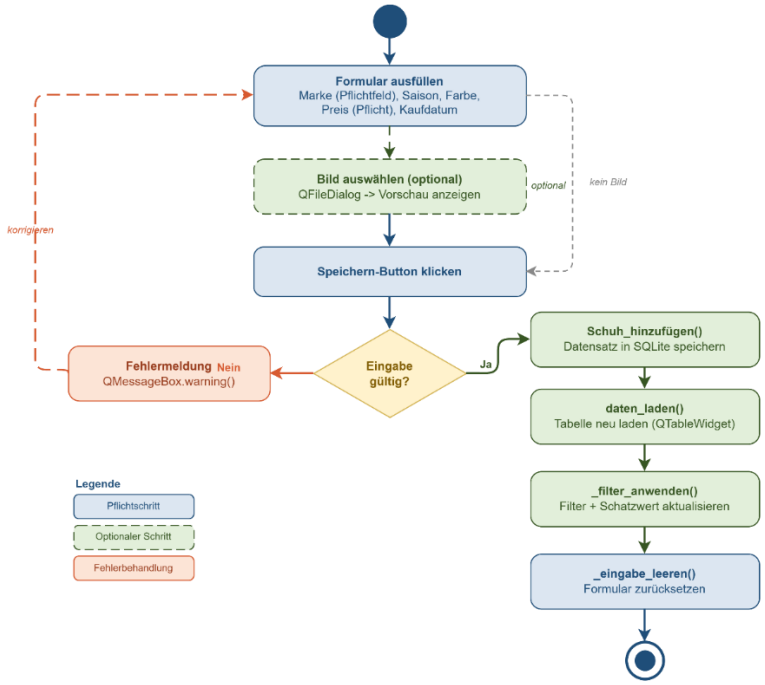


Abbildung 4: Aktivitätsdiagramm - Schuh anlegen (EIS Schuh-Atelier)[18]

11.7 Oberflächenentwürfe: Wireframe, UI-Mockup, Prototyp

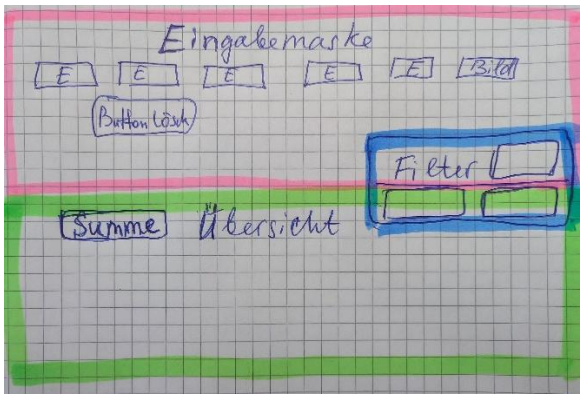


Abbildung 5: Wireframe – Erste grobe Skizze

[18] Zurück zum Kapitel 4.5 Geschäftslogik.

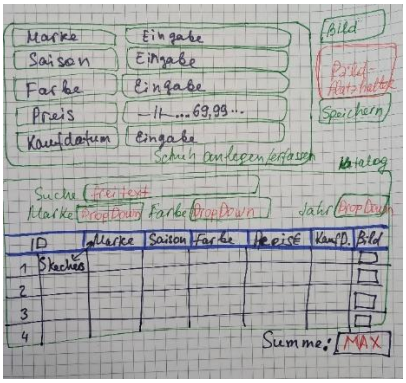


Abbildung 6: Detaillierter Designentwurf final

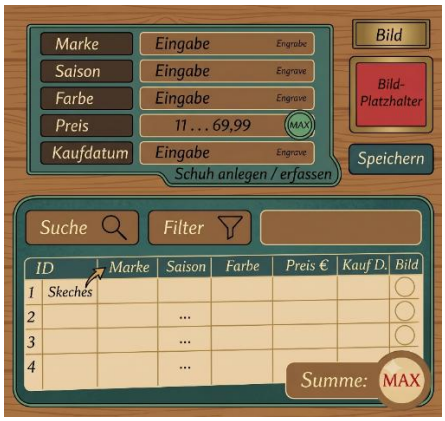


Abbildung 7: Mockup – KI-Visualisierung im Entwurf

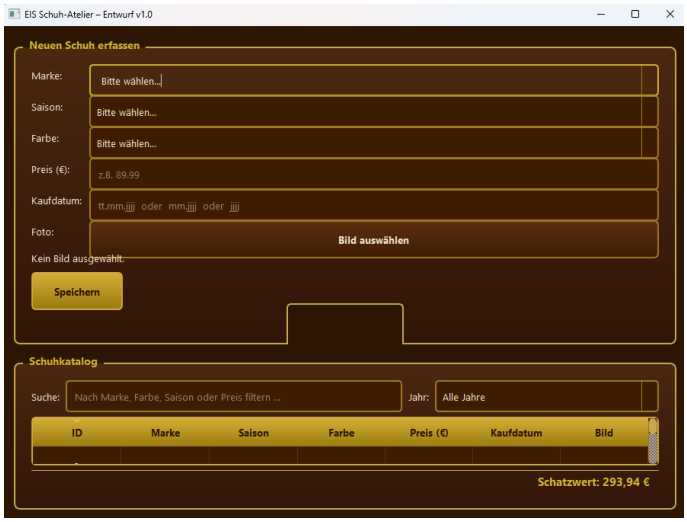


Abbildung 8: Der Prototyp[19]

[19] Zurück zum Kapitel 4.4 Entwurf der Benutzeroberfläche.

11.8 Screenshots der fertigen Anwendung[20]

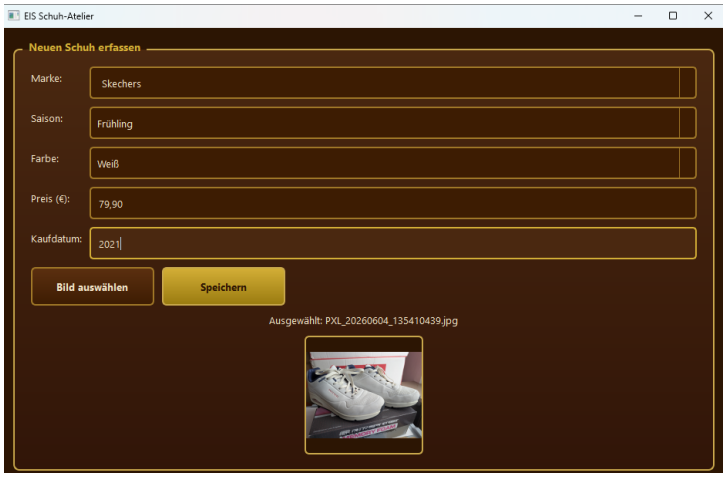


Abbildung 9: Schuh-Datensatz anlegen

Abbildung 10: Schuhkatalog

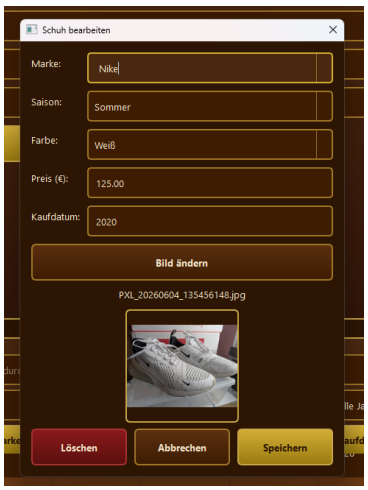


Abbildung 11: Schuhdatensatz bearbeiten / löschen

[20] Zurück zum Kapitel 5.2 Implementierung der Benutzeroberfläche

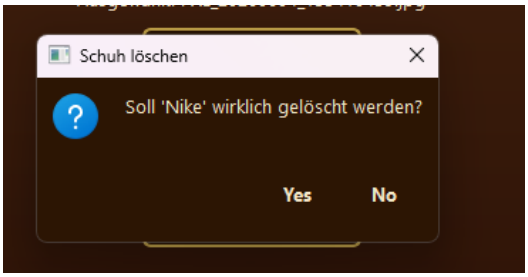


Abbildung 12: Schuhdatensatz löschen

11.9 Entwicklerdokumentation (Auszug)

Die Entwicklerdokumentation basiert auf dem `PEP-257`-Standard für Python-Docstrings. Jede Klasse und jede öffentliche Methode enthält einen einzeiligen oder mehrzeiligen `Docstring`, der Zweck, Parameter und Rückgabewert beschreibt.

Modul `model.py` **— Klasse** `SchuhModell`

```
class SchuhModell:
    """Verwaltet die SQLite-Datenbank und stellt CRUD-Methoden bereit."""

    def schuh_hinzufuegen(self, marke, saison, farbe, preis, kaufdatum, bildpfad):
        """Fügt einen neuen Datensatz in die Tabelle 'schuhe' ein."""

    def alle_schuhe_abrufen(self):
        """Gibt alle Einträge der Tabelle 'schuhe' als Liste zurück."""

    def schuh_aendern(self, schuh_id, marke, saison, farbe, preis, kaufdatum, bildpfad):
        """Aktualisiert alle Felder eines bestehenden Datensatzes anhand der ID."""

    def schuh_loeschen(self, schuh_id):
        """Löscht einen Datensatz anhand der ID unwiderruflich aus der Datenbank."""

    def verbindung_schliessen(self):
        """Schließt die Datenbankverbindung sauber beim Beenden der Anwendung."""
```

Modul `controller.py` **— Klasse** `SchuhController`


```
class SchuhController:
    """Steuert den Datenfluss zwischen SchuhModell und Hauptfens-
ter.
    Wird einmalig von main.py erzeugt. Baut Model und View auf,
    verbindet deren Signale mit den eigenen Methoden (Slots) und
    lädt beim Start alle vorhandenen Daten in die Tabelle.
    """

    def _filter_anwenden(self, *_):
        """Wendet alle vier Filter gleichzeitig mit UND-Logik auf
die Tabelle an.
        Das *_ nimmt PyQt6-Signalparameter (z. B. den neuen Text-
wert bei
        currentTextChanged) entgegen und verwirft sie bewusst –
die Filterung
        liest stets den aktuellen Zustand aller vier Felder selbst
aus.
        Zeilen werden per setRowHidden() aus- bzw. eingeblendet.
        """

    def daten_laden(self):
        """Lädt alle Schuhe aus der Datenbank und befüllt die Ta-
belle."""

    def schuh_speichern(self):
        """Liest das Formular aus, validiert die Eingaben und
speichert den Schuh."""
```

Die vollständige Dokumentation aller Methoden ist direkt im Quellcode in den Dateien `model.py` und `controller.py` hinterlegt.[21]

11.10 Prüf- und Abnahmeprotokoll und Systemtest

Projekt:	EIS Schuh-Atelier	**Prüfdatum:**	Juni 2026
Version:	1.0 (EIS-Schuh-Atelier.exe)	**Testumgebung:**	Windows 11 Pro, Python 3.14
Prüferin:	Ekaterina Eisfeld	**Testmethoden:**	Blackbox-Test, Unit-Test (TDD)
Gesamtergebnis:	**BESTANDEN**	**Offene Punkte:**	Keine

Teil 1: Manuelle Funktionstests (Blackbox)

Die folgenden Testfälle wurden manuell an der kompilierten `.EXE-Datei` durchgeführt. Jeder Testfall wurde mit konkreten Eingabedaten ausgeführt und das tatsächliche Ergebnis mit dem erwarteten Ergebnis verglichen.

Nr.	Testfall	Eingabe / Aktion	Erwartetes Ergebnis	Tatsächliches Ergebnis	Status

[21] Zurück zum Kapitel 8.2 Entwicklerdokumentation.

T01	**Schuh anlegen (vollständig)**	*Marke: Nike, Saison: Sommer, Farbe: Schwarz, Preis: 89,99, Datum: 05.2026, Bild: vorhanden*	Datensatz erscheint in Tabelle. Schatzwert aktualisiert sich.	Eintrag korrekt gespeichert. Schatzwert zeigt 89,99 €.	**Erfolg**
T02	**Schuh anlegen — Pflichtfeld leer**	*Marke: (leer), Preis: 50,00 → Speichern klicken*	Fehlermeldung: 'Bitte gib eine Marke ein'. Kein DB-Eintrag.	`QMessageBox.warning` erscheint. Kein Datensatz gespeichert.	**Erfolg**
T03	**Schuh anlegen — ungültiger Preis**	*Marke: Adidas, Preis: 'abc' → Speichern*	Fehlermeldung: ungültiger Preis. Kein Eintrag.	Fehlermeldung korrekt. Keine DB-Operation.	**Erfolg**
T04	**Schuh anlegen — ungültiges Datum**	*Kaufdatum: '13.13.2026' → Speichern*	Fehlermeldung: ungültiges Datumsformat.	Regex-Validierung schlägt an. Meldung erscheint.	**Erfolg**
T05	**Schuh bearbeiten**	*Klick auf Zeile 1 → Preis ändern auf 120,00 → Speichern*	Tabelle zeigt aktualisierten Preis. Schatzwert passt sich an.	Bearbeitung korrekt gespeichert. Schatzwert aktualisiert.	**Erfolg**
T06	**Schuh löschen mit Bestätigung**	*Klick auf Zeile → Löschen → Bestätigungsdialog → 'Ja'*	Datensatz aus Tabelle und Datenbank entfernt.	Eintrag gelöscht. Schatzwert reduziert sich.	**Erfolg**
T07	**Schuh löschen — Abbruch**	*Klick auf Zeile → Löschen → Bestätigungsdialog → 'Nein'*	Kein Datensatz gelöscht. Tabelle unverändert.	Abbruch funktioniert korrekt. DB unverändert.	**Erfolg**
T08	**Freitextsuche**	*Suchfeld: 'Nike' eingeben*	Nur Nike-Einträge sichtbar. Andere ausgeblendet.	`setRowHidden()` filtert korrekt. Schatzwert passt sich an.	**Erfolg**
T09	**Mehrfach-Filter (UND-Logik)**	*Marke: Adidas, Farbe: Weiß, Jahr: 2025 gleichzeitig aktiv*	Nur Datensätze mit allen 3 Kriterien sichtbar.	UND-Logik korrekt. Schatzwert zeigt gefilterten Wert.	**Erfolg**
T10	**Datenpersistenz (Neustart)**	*2 Schuhe anlegen → App schließen → App neu starten*	Beide Einträge nach Neustart wieder sichtbar.	SQLite-Datenbank korrekt persistiert. Daten vorhanden.	**Erfolg**
T11	**Bild auswählen und anzeigen**	*Bild auswählen → Speichern → Tabellenansicht*	Miniaturansicht in Tabellenspalte. Vorschau im Formular.	`QPixmap` korrekt skaliert und angezeigt.	**Erfolg**
T12	**Schatzwert-Berechnung bei Filter**	*3 Schuhe (50€, 80€, 120€) → Filter auf Marke X (2 Treffer: 50€+80€)*	Gefiltert: 130,00€ | Gesamt: 250,00€	Label zeigt: 'Schatzwert (gefiltert): 130,00 € | Gesamt: 250,00 €'	**Erfolg**

Tabelle 6: Testergebnisse

Teil 2: Automatisierte Unit-Tests[22] KI-Generiert[23]

Die Unit-Tests in `test_model.py` prüfen die Datenbankschicht (`SchuhModell`) unabhängig von der grafischen Oberfläche. Jeder Test verwendet eine isolierte temporäre Datenbankdatei im Hauptverzeichnis (während im Produktivbetrieb die Datei `database/schuhe.db` verwendet wird), die nach jedem Testlauf automatisch gelöscht wird. Die Teststruktur orientiert sich an den drei Entwicklungsstufen (`Red, Green, Refactor`), um die schrittweise Validierung und die anschließende Bereinigung der

[22] (TDD — Red → Green → Refactor)
[23] Siehe Anhang 11.17 KI-Offenlegung.

Codebasis (KISS-Prinzip, Pfadlogik für den EXE-Betrieb) methodisch sauber zu dokumentieren.

Test-ID	Testmethode	Beschreibung	TDD-Phase	Ergebnis
UT-01	test_phase1_red_schuh_hinzufuegen_existiert_noch_nicht	Prüft, ob schuh_hinzufuegen() als Methode existiert (simulierte RED-Phase)	**RED**	**OK**
UT-02	test_phase2_green_schuh_wird_gespeichert	Legt einen Schuh an und prüft, ob genau 1 Datensatz in der DB liegt	**GREEN**	**OK**
UT-03	test_phase3_refactor_alle_felder_korrekt_gespeichert	Prüft alle 5 Felder (marke, saison, farbe, preis, kaufdatum) auf Korrektheit	**REFACTOR**	**OK**

Tabelle 7: Unit-Tests Ergebnisse

Konsolenausgabe: `python -m unittest test_model -v`

```
test_phase1_red_schuh_hinzufuegen_existiert_noch_nicht       (test_model.TestSchuhHinzufuegen)
RED: Test schlägt fehl – schuh_hinzufuegen() war noch nicht implementiert. ... ok
test_phase2_green_schuh_wird_gespeichert   (test_model.TestSchuhHinzufuegen)
GREEN: Minimale Implementierung – genau 1 Eintrag landet in der DB. ... ok
test_phase3_refactor_alle_felder_korrekt_gespeichert         (test_model.TestSchuhHinzufuegen)
REFACTOR: Vollständiger Verhaltenstest – alle Felder werden geprüft. ... ok
----------------------------------------------------------------------
Ran 3 tests in 0.064s
OK
```

Abnahme-Bestätigung

Alle 12 manuellen Funktionstests wurden erfolgreich durchgeführt. Alle 3 automatisierten Unit-Tests wurden ohne Fehler ausgeführt. Sämtliche Muss-Kriterien des Lastenhefts (LH-01 bis LH-09) wurden vollständig und fehlerfrei implementiert.

> **Die Anwendung EIS Schuh-Atelier Version 1.0 wird hiermit offiziell abgenommen. Das Projektziel wurde vollständig erreicht. Die Software ist produktiv einsatzbereit.[24]**

[24] Zurück zum Kapitel 4.6 Maßnahmen zur Qualitätssicherung.

Saarbrücken, 26.06.2026

Ekaterina Eisfeld — Entwicklerin und Auftraggeberin

11.11 Klassendiagramm – MVC Architektur

Die drei Schichten des MVC-Musters sind farblich getrennt: Blau = `SchuhModell`, Grün = `Hauptfenster` / `DetailFenster`, Orange = `SchuhController`. Der Controller kennt Model und View — Model und View kennen sich gegenseitig nicht.

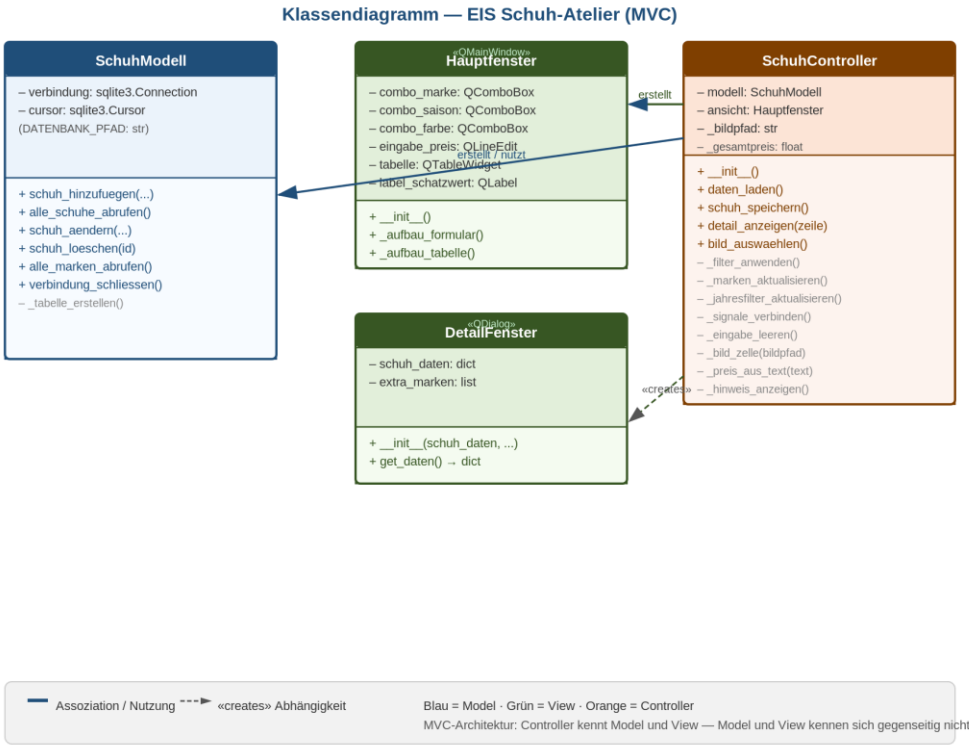


Abbildung 13: Klassendiagramm — MVC-Architektur (model.py / view.py / controller.py)[25]

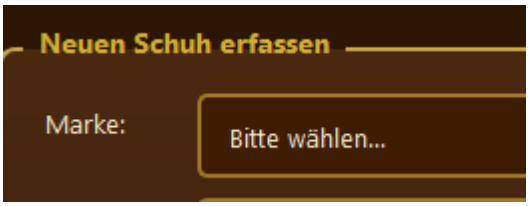
11.12 Benutzerdokumentation (Auszug)

Die komplette Kundendokumentation findet man im Ordner Projektdateien.zip[26]

Im oberen Bereich des `Hauptfensters` findest du das Eingabeformular. So legst du einen neuen Eintrag an:

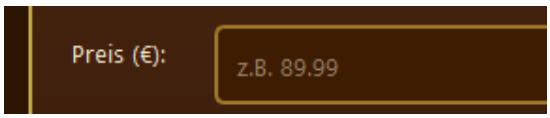
- Wähle die Marke aus dem Dropdown-Menü oder tippe sie direkt ein. Die Marke ist ein Pflichtfeld — ohne eine Angabe kann nicht gespeichert werden.

[25] Zurück zum Kapitel 4.2 Architekturdesign.
[26] Zurück zum Kapitel 8.1 Benutzerdokumentation.



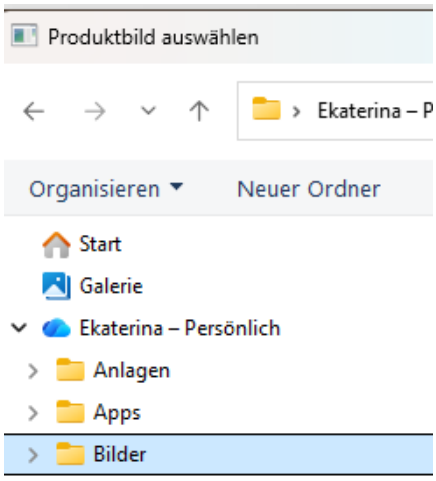
- Wähle optional aus 4 Saisonarten (Sommer, Winter, usw.) und die Farbe aus.

- Trage unbedingt den Kaufpreis ein (z. B. 89,99 oder 89.99 — beides wird akzeptiert).



- Trage optional das Kaufdatum ein. Folgende Formate sind erlaubt:
  - Nur das Jahr: 2024
  - Monat und Jahr: 05.2024
  - Vollständiges Datum: 15.05.2024

- Klicke auf Bild auswählen, um ein Foto von deinem Schuh hinzuzufügen. Wähle eine Bilddatei (JPG, PNG) von deinem Computer aus. Die Vorschau erscheint sofort.



- Klicke auf den goldenen Speichern-Button. Der Schuh erscheint jetzt in der Liste unten.

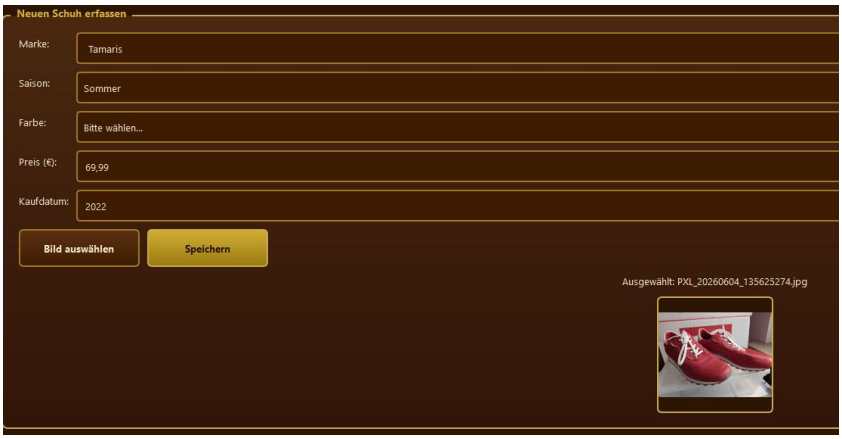


Abbildung 14: Ausgefülltes Formular vor dem Speichern

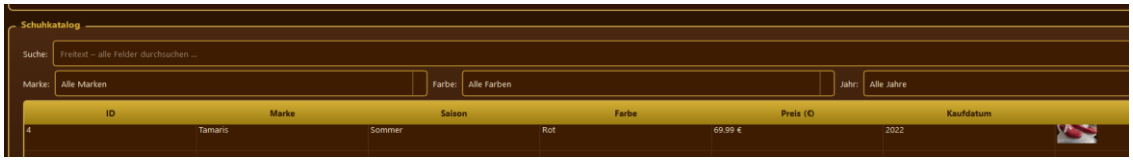


Abbildung 15: Ansicht im Katalog nach dem Speichern

Hinweis: Du kannst eine neue Marke einfach eintippen — sie steht beim nächsten Start automatisch in der Auswahlliste.[27]

11.13 Soll-/Ist-Vergleich

Nr.	Anforderung (Soll)	Ergebnis (Ist)	Bewertung
F1	Schuhe erfassen (Marke, Saison, Farbe, Preis)	Vollstaendig umgesetzt, zusaetzlich Kaufdatum und Produktfoto	☑ Übertroffen
F2	Persistente lokale Datenhaltung	SQLite-Datenbank lokal auf dem Geraet, ohne Cloud-Abhaengigkeit	☑ Erfüllt
F3	Alle CRUD-Operationen (Anlegen, Suchen, Aendern, Loeschen)	PyQt6-Oberflaeche mit Eingabeformular, Live-Suche und Bearbeitungsdialog	☑ Erfüllt

[27] Zurück zum Kapitel 7 Einführungsphase

Nr.	Anforderung (Soll)	Ergebnis (Ist)	Bewertung
F4	Dynamische Filterfunktion	Live-Suche und Jahres-Filter fehlerfrei realisiert	☑ **Erfüllt**
F5	Automatische Berechnung des Investitionswerts	„Schatzwert"-Label mit Echtzeitberechnung, auch bei aktivem Filter	☑ **Übertroffen**
F6	Bildverwaltung mit Vorschau	Bilder werden performant ueber Dateipfade geladen (keine Datenbankbelastung), Miniatur in der Tabelle und Live-Vorschau im Formular	☑ **Übertroffen**

Tabelle 8: Soll-/Ist-Vergleich - Anforderungen

Nr.	Teilaufgabe	Geplant	Tatsächlich	Differenz / Bemerkung
1.1	**Ist-Analyse**	2 h	2 h	*± 0 h*
1.2	**Make-or-Buy-Entscheidung**	2 h	1 h	*- 1 h – Recherche war schneller fertig. Es gibt auf dem Markt kaum solche Anwendungen*
1.3	**Lastenheft**	6	3	*- 3 h – Die Zusammenfassung ging schneller als gedacht.*
2.1	**Architekturdesign (MVC)**	5 h	5 h	*± 0 h*
2.2	**Datenbankmodell**	2 h	2 h	*± 0 h*
2.3	**UI-Mockups**	5 h	10 h	*+ 5 h — zusätzliche Iteration für QSS-Design*
2.4	**Entwicklungsumgebung einrichten**	3 h	3 h	*± 0 h*
3.1	**Model implementieren**	6 h	6 h	*± 0 h*
3.2	**View implementieren**	10 h	10 h	*± 0 h*
3.3	**Controller implementieren**	8 h	8 h	*± 0 h*
3.4	**Integration und Debugging**	4 h	4 h	*± 0 h*
3.5	**PyInstaller-Build**	2 h	2 h	*± 0 h*
4.1	**Funktionstest**	3 h	3 h	*± 0 h*
4.2	**Unit-Tests (TDD)**	3 h	3 h	*± 0 h*
4.3	**Fehlerbehebung**	2 h	1 h	*− 1 h — keine kritischen Fehler*
5.1	**IHK-Dokumentation**	10 h	10 h	*± 0 h*
5.2	**Kundendokumentation**	3 h	3 h	*± 0 h*
5.3	**Anhänge, Abbildungen, Diagramme**	4 h	4 h	*± 0 h*
	Gesamt	**80 h**	**80 h**	*Alle IHK-Vorgaben eingehalten*

Tabelle 9: Soll-/Ist-Vergleich - Zeitplanung[28]

[28] Zurück zum Kapitel 9.1 Soll-/Ist-Vergleich.

11.14 Versionsverwaltung mit Git

1. Zielsetzung

Lokale Git-Versionsverwaltung zur systematischen Sicherung aller Entwicklungsstände mit Möglichkeit zur Wiederherstellung früherer Versionen.

2. Auswahl des Versionsverwaltungssystems

Als Versionsverwaltungssystem wurde Git eingesetzt. Git ist ein verteiltes, kostenloses Open-Source-System, das sich in der professionellen Softwareentwicklung als Industriestandard etabliert hat. Für dieses Projekt sprachen folgende Gründe:

Kriterium	Begründung
Lokale Nutzung	Keine Internetverbindung erforderlich; alle Daten liegen lokal auf dem Entwicklungsrechner
Nachvollziehbarkeit	Jeder Entwicklungsstand wird mit Zeitstempel und Beschreibung gespeichert
Rückfallsicherheit	Ältere Versionen können jederzeit wiederhergestellt werden
Kostenlos	Keine Lizenzkosten, sofort einsatzbereit
Branchenstandard	Weitverbreitetes Wissen, langfristig wartbar

Tabelle 10: Git Vorteile

3. Initialisierung des Repositories

Das Git-Repository wurde im Hauptverzeichnis des Projekts angelegt:

`C:\Users\eeisf\eis_schuh_atelier\`

Die Initialisierung erfolgte mit dem Befehl:

`git init`

Git legt daraufhin automatisch einen versteckten Unterordner .git\ an, in dem alle Versionsdaten intern gespeichert werden. Dieser Ordner wird nicht manuell bearbeitet.

4. Konfiguration der .gitignore-Datei

Nicht alle Dateien eines Projekts eignen sich für die Versionsverwaltung. Automatisch erzeugte Dateien, kompilierte Artefakte und persönliche Daten werden bewusst ausgeschlossen, um das Repository schlank und übersichtlich zu halten. Die .gitignore-Datei enthält folgende Ausschlussregeln:

Pfad / Muster	Begründung
`__pycache__/ *.pyc *.pyo`	Automatisch generierter Bytecode — nicht versionierungsrelevant
`build/ dist/`	PyInstaller-Artefakte — reproduzierbar aus Quellcode
`database/schuhe.db`	Laufzeitdaten, datenschutzrelevant
`logs/`	Laufzeitprotokolle — irrelevant für Reproduzierbarkeit

Tabelle 11: Ausschlussregeln

5. Verwaltete Quelldateien

In das Repository wurden ausschließlich die selbst erstellten Quelldateien aufgenommen, die den eigentlichen Entwicklungsstand des Projekts abbilden:

Datei	Beschreibung
`main.py`	Einstiegspunkt der Anwendung — startet `QApplication` und `Controller`
`model.py`	Datenbankschicht — liest und schreibt Schuheinträge in SQLite
`view.py`	Präsentationsschicht — PyQt6-Benutzeroberfläche (`Hauptfenster`, DetailFenster)
`controller.py`	Steuerungsschicht — verbindet Model und View nach MVC-Muster
`test_model.py`	Unit-Tests nach TDD-Methode (Red → Green → Refactor)
`EIS-Schuh-Atelier.spec`	Konfigurationsdatei für `PyInstaller` zum Erstellen der `.exe`
`.gitignore`	Ausschlussliste für die Versionsverwaltung

Tabelle 12: Übersicht der Quelldateien in Git

6. Commit-Verlauf (git log --oneline)

Im Laufe des Projekts entstanden folgende Commits, die den Entwicklungsprozess lückenlos und chronologisch dokumentieren:

```
7ee81fd TDD-Zyklus vollständig dokumentiert (Red → Green → Refactor)
c798473 Kommentare für IHK-Dokumentation ergänzt
3ef2a5b Code-Cleanup: Redundanzen entfernt, KISS-Prinzip angewendet
82a4d91 Datenbank-Migration für Attribut 'kaufdatum' implementiert
b6d3f8b Unit-Test für schuh_hinzufuegen() nach TDD-Methode
f40c093 Schatzwert: Gesamtwert immer sichtbar, bei Filter zusätzlich
aafba80 Mehrattribut-Filterung hinzugefügt (Marke + Farbe + Jahr)
1660eb5 Schuh-Löschfunktion hinzugefügt
d2c896c Initial Commit: EIS Schuh-Atelier v1.0
```

7. Workflow für künftige Änderungen

Für jede zukünftige Änderung am Quellcode wird folgender Workflow angewendet:

#	Schritt	Befehl / Aktion
1	**Änderung durchführen**	Quellcode bearbeiten (z. B. Fehlerbehebung, neues Feature)
2	**Stagen**	`git add <dateiname>` — geänderte Datei für Commit vormerken
3	**Commit**	`git commit -m "Beschreibung"` — Stand dauerhaft sichern
4	**Verlauf prüfen**	`git log --oneline` — alle Commits einsehen

Tabelle 13: Workflows für künftige Änderungen

Durch diesen Workflow entsteht eine lückenlose, chronologische Geschichte aller Entwicklungsstände. Jeder Commit ist mit Datum, Uhrzeit und Beschreibung versehen und kann bei Bedarf vollständig wiederhergestellt werden.

11.15 Datenverarbeitungskonzept

Datenfluss im MVC-Muster

Die Datenverarbeitung folgt strikt dem MVC-Entwurfsmuster. Jede Schicht hat eine klar abgegrenzte Verantwortung:

```
Benutzer
   │
   ▼
View (view.py)              ← Eingabe: Formularfelder, Buttons
   │  Signal (clicked / textChanged)
   ▼
Controller (controller.py) ← Validierung, Steuerung
   │  Methodenaufruf
   ▼
Model (model.py)            ← SQL-Abfragen, Datenbankzugriff
   │  Rückgabe (sqlite3.Row)
   ▼
Controller                  ← Aufbereitung der Daten
   │  Tabelle befüllen, Label aktualisieren
   ▼
View                        ← Anzeige: Tabelle, Schatzwert-Label
```

Model und View kennen sich gegenseitig nicht. Der Controller ist die einzige Komponente, die beide kennt und zwischen ihnen vermittelt.

Datenbankoperationen (CRUD)

Alle Datenbankzugriffe erfolgen über die Klasse `SchuhModell` in `model.py`. Für den Spaltenzugriff wird `sqlite3.Row` als `row_factory` verwendet, sodass Felder per Name (z. B. `zeile["marke"]`) statt per Index abgerufen werden können.

Operation	Methode	SQL-Befehl
Anlegen	`schuh_hinzufuegen()`	`INSERT INTO schuhe (...) VALUES (?, ?, ?, ?, ?, ?)`
Lesen	`alle_schuhe_abrufen()`	`SELECT … FROM schuhe`
Ändern	`schuh_aendern()`	`UPDATE schuhe SET … WHERE id=?`
Löschen	`schuh_loeschen()`	`DELETE FROM schuhe WHERE id=?`
Hilfsdaten	`alle_marken_abrufen()`	`SELECT DISTINCT marke FROM schuhe ORDER BY marke`

Tabelle 14: Datenbankoperationen (CRUD)

Jede schreibende Operation wird unmittelbar mit `commit()` in die Datenbankdatei geschrieben, sodass kein Datenverlust durch unerwartete Programmbeendigungen entsteht.

Eingabevalidierung

Vor jeder Speicheroperation prüft der Controller die Benutzereingaben. Ungültige Eingaben werden mit einer Fehlermeldung zurückgewiesen; es findet kein Datenbankzugriff statt.

Feld	Validierungsregeln
Marke	Pflichtfeld — darf nicht leer sein und nicht dem Platzhaltertext entsprechen
Preis	Muss in eine Dezimalzahl konvertierbar sein; Komma wird als Dezimaltrennzeichen akzeptiert
Kaufdatum	Muss leer sein oder einem der drei erlaubten Formate entsprechen (`tt.mm.jjjj`, `mm.jjjj`, `jjjj`) — geprüft per regulärem Ausdruck

Tabelle 15: Validierungsregeln

Filterfunktion und Schatzwertberechnung

Die Filterfunktion arbeitet rein auf der bereits geladenen Tabellenansicht, ohne erneute Datenbankabfragen auszulösen. Zeilen, die nicht allen aktiven Filterkriterien entsprechen, werden per `setRowHidden()` ausgeblendet.

Es sind vier Filter gleichzeitig kombinierbar (UND-Logik):

- **Freitext:** durchsucht alle sichtbaren Spalten einer Zeile
- **Markenfilter:** exakter Abgleich mit Spalte „Marke“
- **Farbfilter:** exakter Abgleich mit Spalte „Farbe“
- **Jahresfilter:** Abgleich des Jahresanteils im Kaufdatum

Nach jeder Filteränderung wird der Schatzwert neu berechnet: Ohne aktiven Filter zeigt das Label den Gesamtwert aller Einträge; bei aktivem Filter werden gefilterter Wert und Gesamtwert nebeneinander ausgegeben.

Datenpersistenz und Speicherort

Die Datenbankdatei `database/schuhe.db` wird relativ zum Speicherort der ausführbaren Datei abgelegt. Im kompilierten Betrieb (`.exe` via PyInstaller) wird dafür `sys.executable` ausgewertet, da PyInstaller ein temporäres Entpackverzeichnis verwendet, das sich bei jedem Start ändert. Dadurch bleiben die Daten auch nach Updates oder Neuinstallationen erhalten, solange die Datenbankdatei neben der `.exe` verbleibt.

Beim Beenden der Anwendung wird die Datenbankverbindung über das `aboutToQuit`-Signal der `QApplication` sauber geschlossen, um Dateikorruption zu vermeiden.[29]

[29] Zurück zum Kapitel 5.1 Implementierung der Datenstruktur.

11.16 Screenshots für Implementierung der Geschäftslogik

```
def _signale_verbinden(self):
    self.ansicht.btn_bild.clicked.connect(self.bild_auswaehlen)
    self.ansicht.btn_speichern.clicked.connect(self.schuh_speichern)
    self.ansicht.tabelle.cellClicked.connect(self.detail_anzeigen)
    # Alle vier Filter lösen dieselbe Methode aus.
    self.ansicht.eingabe_suche.textChanged.connect(self._filter_anwenden)
    self.ansicht.combo_filter_marke.currentTextChanged.connect(self._filter_anwenden)
    self.ansicht.combo_filter_farbe.currentTextChanged.connect(self._filter_anwenden)
    self.ansicht.combo_jahr.currentTextChanged.connect(self._filter_anwenden)
```

Abbildung 16: Screenshot für signale_verbinden

```
def _filter_anwenden(self, *_argumente):
    suchtext        = self.ansicht.eingabe_suche.text().strip().lower()
    marke_auswahl   = self.ansicht.combo_filter_marke.currentText()
    farbe_auswahl   = self.ansicht.combo_filter_farbe.currentText()
    jahr_auswahl    = self.ansicht.combo_jahr.currentText()

    for zeile in range(tabelle.rowCount()):
        freitext_trifft_zu = not suchtext or any(…)
        marke_trifft_zu    = not aktiver_marken_filter or …
        farbe_trifft_zu    = not aktiver_farben_filter or …
        jahr_trifft_zu     = not aktiver_jahres_filter or …

        zeile_ausblenden = not (freitext_trifft_zu and marke_trifft_zu
                                and farbe_trifft_zu and jahr_trifft_zu)
        tabelle.setRowHidden(zeile, zeile_ausblenden)
```

Abbildung 17: Screenshot für filter_anwenden

```
def schuh_speichern(self):
    # Platzhalter normalisieren
    if saison == PLATZHALTER:
        saison = ""
    if farbe == PLATZHALTER:
        farbe = ""

    if not marke.strip() or marke == PLATZHALTER:
        self._hinweis_anzeigen("Pflichtfeld fehlt", "Bitte gib eine Marke ein …")
        return

    if kaufdatum and not _DATUM_MUSTER.match(kaufdatum):
        self._hinweis_anzeigen("Ungültiges Kaufdatum", …)
        return

    try:
        preis = float(preis_text.replace(",", "."))
    except ValueError:
        self._hinweis_anzeigen("Ungültiger Preis", …)
        return
```

Abbildung 18: Screenshot für schuh_speichern[30]

[30] Zurück zum Kapitel 5.3 Implementierung der Geschäftslogik.

11.17 KI-Offenlegung

Im Rahmen der Erstellung dieser Projektdokumentation sowie während der Implementierungsphase wurden KI-Assistenzsysteme unterstützend eingesetzt. Die nachfolgende Aufstellung legt offen, welche Werkzeuge verwendet wurden und in welchem Umfang.

Eingesetzte KI-Werkzeuge

Werkzeug	Anbieter	Einsatzbereich
Claude	Anthropic	Formulierungshilfe, Strukturierung von Textabschnitten, Konsistenzprüfung der Dokumentation, Code-Vervollständigung während der Implementierungsphase, Unit-Tests codieren und durchführen.
ChatGPT	OpenAI	Recherche, Erklärung technischer Konzepte, Textüberarbeitung
Gemini	Google	Ergänzende Recherche und Formulierungsvorschläge
Microsoft Copilot	Microsoft	Recherche, Formulierungsvorschläge und Textüberarbeitung

Tabelle 16: KI-Werkzeuge

Art und Umfang der Nutzung

Die KI-Werkzeuge wurden ausschließlich als Hilfsmittel eingesetzt. Sämtliche fachlichen Inhalte, Architekturentscheidungen, Implementierungen und technischen Konzepte stammen aus eigener Arbeit. Kein Text und kein Code wurden unverändert übernommen.

KI wurde konkret eingesetzt für:

- Verbesserung von Formulierungen in deutschen Fachtexten
- Erklärung technischer Zusammenhänge (PyQt6, SQLite, MVC)
- Strukturvorschläge für einzelne Dokumentationsabschnitte
- Unterstützung bei Recherche und Textformulierung (Microsoft Copilot)
- Generierung des vollständigen Unit-Test-Codes (`test_model.py`) im Rahmen der TDD-Qualitätssicherung.[31]

Der Quellcode der Kernanwendung (`model.py, view.py, controller.py, main.py`) wurde eigenständig entwickelt, verstanden und getestet. Der Testcode (`test_model.py`) wurde mit Unterstützung von KI-Assistenzsystemen generiert und anschließend eigenständig geprüft und eingebunden. Die Verantwortung für alle Inhalte dieser Dokumentation liegt vollständig bei der Verfasserin.

Ekaterina Eisfeld

11.18 Code

[31] Zurück zum Kapitel 5.5 Implementierung der automatisierten Unit-Tests (TDD).