**IU Internationale Hochschule**

Studiengang: Informatik M.Sc.

**Phase 3: Finalisierungsphase**

Portfolio zur Prüfung im Kurs Projekt: Software Engineering (DLMCSPSE01\_D)

eingereicht von: Kevin Walter

Matrikel-Nr.: 92212082

Tutor: Prof. Dr.-Ing. David Kuhlen

Datum: 19.08.2025

Inhaltsverzeichnis

[Phase 1: Konzeptionsphase – Projektdokument 1](#_Toc204855238)

[1. Projektidee und angestrebtes Ergebnis 1](#_Toc204855239)

[2. Potenzielle Risiken und Gegenmaßnahmen 1](#_Toc204855240)

[2.1 Skala: 1](#_Toc204855241)

[2.2 Tabelle potenzielle Risiken und Gegenmaßnahmen 1](#_Toc204855242)

[3. Projektstrukturplan (PSP) 2](#_Toc204855243)

[4. Gantt-Diagramm 3](#_Toc204855244)

[Phase 1: Konzeptionsphase - Anforderungsdokument 4](#_Toc204855245)

[5. Stakeholder (Ziel- und Benutzergruppe) 4](#_Toc204855246)

[6. Funktionale Anforderungen 4](#_Toc204855247)

[6.1 Funktionsliste 4](#_Toc204855248)

[6.2 User Stories 4](#_Toc204855249)

[7. Nicht-funktionale Anforderungen 5](#_Toc204855250)

[8. Konzept zur Qualitätssicherung 6](#_Toc204855251)

[9. Glossar 6](#_Toc204855252)

[Phase 1: Konzeptionsphase - Spezifikationsdokument 7](#_Toc204855253)

[10. Datenmodell 7](#_Toc204855254)

[10.1 Aufgabe 7](#_Toc204855255)

[10.2 MVC-Muster und Stellungnahme zum Einsatz 7](#_Toc204855256)

[10.3 UML-Klassendiagramm 8](#_Toc204855257)

[10.4 Persistenzschicht 7](#_Toc204855258)

[10.5 Nachträgliche Ergänzungen und Änderungen 9](#_Toc204855259)

[11. Geschäftsprozesse 10](#_Toc204855260)

[12. Geschäftsregeln 12](#_Toc204855261)

[13. Systemschnittstellen 12](#_Toc204855262)

[14. Benutzerschnittstellen 12](#_Toc204855263)

[14.1 Struktur der Oberfläche 12](#_Toc204855264)

[14.2 Wichtigste Dialoge & Abläufe 12](#_Toc204855265)

[14.3 Skizze der Anwendung 13](#_Toc204855266)

[15. Systemkontext und Datenflüsse 14](#_Toc204855267)

[15.1 Anwendungsspezifischer Kontext 14](#_Toc204855268)

[15.2 Abgrenzung und Kontextdarstellung 14](#_Toc204855269)

[Phase 2: Erarbeitungs- und Reflexionsphase – Architekturdokument 15](#_Toc204855270)

[16. Technologieübersicht 15](#_Toc204855271)

[16.1 Programmiersprache 15](#_Toc204855272)

[16.2 Frameworks 15](#_Toc204855273)

[16.3 Bibliotheken 15](#_Toc204855274)

[16.4 Entwicklungswerkzeuge 16](#_Toc204855275)

[17. Architekturübersicht 16](#_Toc204855276)

[18. Struktur 17](#_Toc204855277)

[18.1 Hauptkomponenten 17](#_Toc204855278)

[18.2 Abhängigkeiten 19](#_Toc204855279)

[18.3 Erweiterbarkeit 20](#_Toc204855280)

[19. Verhalten 21](#_Toc204855281)

[Quellenverzeichnis 33](#_Toc204855282)

# Phase 1: Konzeptionsphase – Projektdokument

Link zum GitHub Repository: <https://github.com/KTWIU/SEProject>

# Projektidee und angestrebtes Ergebnis

Zu Beginn wird ein alltägliches Problem beschrieben. Die Tagesplanung erfolgt häufig auf einem klassischen Notizblock, dieses Vorgehen ist auf Dauer unpraktisch und ressourcenintensiv. Zwar existiert die Erinnerungen-App für das iPhone, jedoch fehlt eine einfache und schlanke Anwendung für den PC, die die Planung im Alltag unterstützt. Viele digitale Tools sind entweder zu komplex oder bieten unnötig viele Zusatzfunktionen.

Zielsetzung dieses Portfolios ist die Entwicklung einer schlanken Desktop-Anwendung für Windows-PCs, mit der Aufgaben für den kommenden Tag oder die gesamte Woche effizient und übersichtlich geplant werden können. Die Benutzeroberfläche ist bewusst einfach gehalten, ohne überflüssige Menüs oder Reiter, mit klarem Fokus auf die tägliche Aufgabenplanung, Deadlines und eine Kalenderansicht.

# Potenzielle Risiken und Gegenmaßnahmen

## 2.1 Skala

|  |  |  |  |
| --- | --- | --- | --- |
| Wahrscheinlichkeit | niedrig | mittel | hoch |
| Schadensausmaß | niedrig | mittel | hoch |

## 2.2 Tabelle potenzielle Risiken und Gegenmaßnahmen

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Risiko | Eintrittswahrscheinlichkeit | Frühwarnindikator | Schadensausmaß | Gegenmaßnahme |
| Einarbeitung dauert zu lange | mittel | Erste GUI-Tests  >2 Tage verspätet | mittel | Früh mit Einarbeitung beginnen, Beispiele als Orientierung verwenden |
| Release-Package funktioniert nicht auf anderen Rechnern | hoch | Anwendung startet nicht auf Test-PC | hoch | windeployqt verwenden,  damit alle DLLs und Dateien vorhanden sind |
| Projektumfang zu groß /  Zeitmangel | gering | Aufgabenrückstand in Woche 2 | mittel | Realistischer Zeitplan,  Fokus auf Kernfunktionen  (sh. Abschnitt 1) |
| Fehler durch Pointer /  Referenzen | mittel | Wiederholte Abstürze oder Warnungen | hoch | Einfacher Code, Debugging-Tools und Testing |
| Codeverlust /  verschiedene Versionen | gering | Mehrere lokale Kopien auf Laufwerk | hoch | Regelmäßige Pushs zu GitHub (GitHub, Version 2.46.0.windows.1) |

# Projektstrukturplan (PSP)



Quelle: Eigene Darstellung mit Microsoft Visio (Microsoft 365, Version 2506)

# Gantt-Diagramm



Quelle: Eigene Darstellung mit Microsoft Excel (Microsoft 365, Version 2506)

# Phase 1: Konzeptionsphase - Anforderungsdokument

# Stakeholder (Ziel- und Benutzergruppe)

Die Anwendung richtet sich an alle Nutzer, die ihren Tag gerne strukturiert planen möchten, unabhängig davon ob sie Studenten, Berufstätige oder Privatpersonen sind. Die Anwendung stellt eine digitale Alternative zum analogen Notizblock dar und dient der übersichtlichen Verwaltung anstehender Aufgaben sowie Fälligkeitstermine. Das Ziel ist es, die Vorteile moderner Aufgabenverwaltung am PC mit der Einfachheit eines Papier-Notizblocks zu verbinden, um papierlos und effizienter planen zu können.

# Funktionale Anforderungen

Im folgenden Kapitel werden die funktionalen Anforderungen der Anwendung aufgezeigt, außerdem wird in Abbildung 1 ein Use-Case Diagramm dargestellt.

## 6.1 Funktionsliste

* Aufgaben anlegen, bearbeiten, löschen
* Deadlines setzen
* Aufgaben als erledigt markieren
* Kalender-/Tagesansicht
* Daten speichern/laden

## 6.2 User Stories

Als Benutzer möchte ich einfach per Texteingabe mit der Tastatur Aufgaben anlegen, bearbeiten und löschen können.

Als Benutzer möchte ich für meine Aufgabe auch ein Fälligkeitsdatum setzen können, damit ich mir nicht selbst notieren muss, bis wann ich die Aufgabe erledigen will. Es ist vorerst nicht vorgesehen, eine Benachrichtigung zu versenden, dass die Aufgabe überfällig ist. Über das heutige Datum und dem notierten Fälligkeitsdatum kann ich mir das selbst herleiten.

Als Benutzer möchte ich die Aufgabe als erledigt markieren können, damit ich direkt sehen kann, welche Aufgaben ich heute noch vor habe.

Als Benutzer möchte ich eine Kalender- und Tagesansicht sehen können, damit ich so beispielsweise auch eine ganze Woche planen kann.

Als Benutzer möchte ich die Daten speichern können, das heißt entweder per „save“ Button, oder mit automatischer Speicherung.

Als Benutzer muss ich mich nicht anmelden oder registrieren. Die Anwendung funktioniert wie ein klassisches Notizbuch: Ich kann direkt starten, ohne Account oder Internetverbindung. Die Aufgaben werden lokal auf meinem Windows-PC gespeichert. Eine Synchronisierung mit einer Cloud ist nicht vorgesehen.

Abbildung 1: Use-Case-Diagramm

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

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung mit Microsoft Visio (Microsoft 365, Version 2506)

# Nicht-funktionale Anforderungen

|  |  |
| --- | --- |
| Nicht-funktionale Anforderung | Beschreibung und Erfüllbarkeit |
| Benutzerfreundlichkeit | Die Anwendung besitzt nur ein Hauptfenster und maximal 2 Dialoge. Alle Funktionen sind mit maximal 2 Klicks erreichbar. |
| Performance | Startzeit < 2 Sekunden auf einem Windows-PC mit SSD. Alle Listen reagieren ohne spürbare Verzögerung. |
| Datensicherheit | Alle Daten werden lokal auf dem PC abgespeichert. Keine Verbindung zu einem Netzwerk oder Cloud-Service. |
| Plattform | Die Anwendung läuft unter Windows 10/11 und wurde mit Qt getestet. Kein plattformspezifischer Code. |
| Eingabevalidierung | Pflichtfeldprüfung (Titel darf nicht leer sein) und Fälligkeitsdatum darf nicht in der Vergangenheit liegen. |
| Verschlüsselung | Nicht vorgesehen, da die Daten lokal und ohne Internetzugriff gespeichert werden. |

# Konzept zur Qualitätssicherung

Zur Sicherstellung der Softwarequalität werden drei Teststufen geplant:

* Unit-Tests: Um die Kernfunktionen der Geschäftslogiksicherzustellen (z.B. TaskManager-Methoden) werden automatisierte Unit-Tests geschrieben. Diese Tests werden mit Hilfe von googletest (googletest, Version 1.17.0) automatisiert und decken zum Beispiel das Hinzufügen, Löschen und Bearbeiten von Aufgaben ab.
* Integrationstests: Um das Zusammenspiel zwischen den einzelnen Komponenten (z.B. zwischen Controller und View) zu testen, werden Integrationstests geschrieben. Auch die Integrationstests sollen so weit wie möglich automatisiert werden, ist das nicht Möglich werden die Tests durch Screenshots im Kapitel Qualitätssicherung in Phase 3 dokumentiert.
* Systemtests: Die gesamte Anwendung wird unter realen Einsatzbedingungen manuell getestet. Hier werden typische Benutzeraktionen über die grafische Benutzeroberfläche geprüft (z.B. Aufgabenverwaltung, Speichern/Laden). Eine vollständige Automatisierung dieser Tests ist nicht vorgesehen.

Das Ziel der Tests ist es, die Zuverlässigkeit der Anwendung auf jeder Ebene sicherzustellen und Fehler frühzeitig zu erkennen.

# Glossar

* **Aufgabe:** Ein Eintrag mit Titel, Beschreibung und Fälligkeitsdatum
* **Tagesansicht:** Auflistung aller an diesem Tag zu erledigenden/eingetragenen Aufgaben
* **Deadline:** Datum, bis zu dem die Aufgabe abgeschlossen sein soll
* **GUI:** Grafische Benutzeroberfläche
* **User** **Story:** Beschreibung aus Anwendersicht
* **UML:** Unified Modeling Language

# Phase 1: Konzeptionsphase - Spezifikationsdokument

# Datenmodell

## 10.1 Aufgabe

* Es ist das wichtigste Objekt
* Attribute einer Aufgabe sind: Titel, Beschreibung, Fälligkeitsdatum, Status (offen/erledigt) mit Anzeigen in grün/rot
* Benötigt mindestens einen Titel (Pflichtfeld), ein optionales Fälligkeitsdatum und einen Status (offen/erledigt), kann zusätzliche Beschreibung haben

## 10.2 MVC-Muster und Stellungnahme zum Einsatz

Das Model-View-Controller (MVC) Muster wird zur Trennung von Präsentation, Steuerung und Datenmodell verwendet. Es wird eingesetzt, wenn unterschiedliche Schichten auf die Daten notwendig sind oder zukünftige Anforderungen an die Darstellung und Interaktion noch nicht feststehen. Das Modell übernimmt die Datenhaltung, die View stellt die Darstellung (GUI) sicher und der Controller vermittelt zwischen den beiden Instanzen, indem er die Benutzereingabe verarbeitet und Veränderungen im Modell anstößt. Diese Trennung erhöht die Wartbarkeit und Erweiterbarkeit der Anwendung (Sommerville, 2012, S. 191–193).

Für die Umsetzung konkret dieser Anwendung wird das MVC-Muster wie in der gängigen Literatur bekannt als Rahmen gewählt. Die drei Schichten View (MainWindow), Model (Task, TaskManager) und Controller (TaskController) werden strikt voneinander getrennt. Zur besseren Entkopplung ist außerdem ein Interface (TaskInterface) vorgesehen. Zusätzlich ist geplant, auch für Controller (TaskControllerInterface) und View (ViewInterface) eigene Interfaces einzuführen, um die Modularität und Testbarkeit der Software weiter zu erhöhen. Mit dieser Architektur ist es dann auch leichter möglich, die Anwendung über dieses Projekt hinaus weiterzuentwickeln (z.B. Netzwerkschnittstelle).

## 10.3 Factory Pattern (TaskFactory)

Das Factory-Pattern ist ein Entwurfsmuster, das die Erzeugung von Objekten kapselt und zentralisiert. Anstatt überall im Code direkt neue Objekte zu erstellen, übernimmt eine spezielle Factory-Klasse (TaskFactory) diese Aufgabe (Balzert, 2009, S. 131).

Die Klasse TaskFactory erzeugt alle Task-Objekte. Die Factory wird sowohl beim Hinzufügen einer neuen Aufgabe durch den Nutzer als auch beim Laden von Aufgaben aus der Datei verwendet. So bleibt die Objekt-Erzeugung an einer Stelle gebündelt und kann bei Änderungen leicht angepasst werden.

## 10.4 Persistenzschicht

Damit Aufgaben dauerhaft, also auch über die Sitzung hinaus gespeichert werden können, verfügt die Anwendung über eine Persistenzschicht. Die Aufgaben werden lokal in einer Datendatei gespeichert und beim Start der Anwendung wieder eingelesen (Rau & Schuster, 2021, S. 58). Diese Vorgehen erfüllt die Anforderungen an eine Persistenzschicht, wie sie auch mit einer Datenbank realisierbar wäre. Die Architektur ermöglicht es, die Persistenzschicht später problemlos durch eine alternative Implementierung wie einer Datenbank zu ersetzen.

Die Persistenz erfolgt durch Speicherung aller Aufgaben in einer lokalen CSV-Datei namens TaskFile.csv. Die Datei besteht aus einer Kopfzeile und einer beliebigen Anzahl von Datenzeilen. Jede Zeile entspricht einer Aufgabe. Die Werte werden durch ein Semikolon getrennt, die Struktur der Datei sieht dann wie folgt aus:

* Index;Titel;Beschreibung;Faelligkeitsdatum;Erledigt;

Mit folgender Bedeutung:

* Index: Eindeutiger numerischer Wert je Aufgabe
* Titel: Titel der Aufgabe
* Beschreibung: Beschreibungstext
* Faelligkeitsdatum: Datum als String (Format YYYY-MM-DD)
* Erledigt: Status der Aufgabe (true/false)

Beim Laden werden die Werte zeilenweise eingelesen, durch das Semikolon getrennt und den Attributen der Aufgaben-Objekte zugewiesen. Beim Speichern werden alle aktuellen Aufgaben in derselben Reihenfolge zurück in die Datei geschrieben.

## 10.5 Beans

Das JavaBeans-Muster sieht vor, dass zur Objekterzeugung ein parameterloser Konstruktor verwendet wird und die Eigenschaften anschließend über Set-Methoden gesetzt werden (Boch, 2018, S. 12–13). Darüber hinaus ist es üblich, alle Attribute privat zu deklarieren und ausschließlich über öffentliche Getter- und Setter-Methoden darauf zuzugreifen, um die Datenkapselung zu gewährleisten (Silberbauer, 2020, S. 51–52).

Dieses Prinzip wurde in der Klasse Task übernommen, alle Attribute der Klasse sind als private deklariert und der Zugriff erfolgt ausschließlich über öffentliche Getter- und Setter-Methoden. Ein parameterloser Standard-Konstruktor ist vorhanden, sodass Aufgabenobjekte wie beim Bean-Prinzip erstellt und anschließend über Setter individuell befüllt werden können. Damit entspricht die Implementierung von Task dem JavaBeans-Konzept und gewährleistet eine saubere Datenkapselung.

## 10.6 UML-Klassendiagramm

Die aus dem vorherigen Abschnitt zum MVC-Muster gewonnen Erkenntnisse werden im UML-Klassendiagramm in Abbildung 2 angewendet.

Abbildung 2: UML-Klassendiagramm



Quelle: Eigene Darstellung mit Microsoft Visio (Microsoft 365, Version 2506)

## 10.7 Nachträgliche Ergänzungen und Änderungen

Wie bereits kurz in Abschnitt 9.2 erwähnt, ist es mit dem MVC-Muster leicht möglich, nachträgliche Ergänzungen und Änderungen einzubauen. Durch das MVC-Muster und die Einführung des Interfaces zwischen Controller und Model ist es möglich, neue Funktionen (z.B. alternative Speicherformate, Erweiterungen in der GUI oder zusätzliche Logik) einzuführen ohne tiefgreifende Änderungen an den bestehenden Komponenten vorzunehmen. Das Interface ermöglicht es außerdem, die Datenhaltung mit minimalen Anpassungsaufwand zu ersetzen. So kann beispielsweise im Nachhinein auf eine Datenbank oder Netzwerkspeicherung umgestiegen werden.

# Geschäftsprozesse

**Geschäftsprozess 1: Aufgabe anlegen**

1. Der Benutzer startet die Anwendung
2. Im Hauptfenster klickt er auf „Aufgabe hinzufügen“
3. Es öffnet sich ein Modal
   * Der Benutzer gibt Titel, Beschreibung, Fälligkeitsdatum ein
   * Mit „Speichern“ wird die Aufgabe übernommen
4. Die neue Aufgabe erschein in der Aufgabenliste

**Geschäftsprozess 2: Aufgabe als erledigt markieren**

1. Benutzer sieht die Aufgabenliste
2. Bei einer Aufgabe klickt er auf „Aufgabe als erledigt markieren“
3. Die Aufgabe wird als erledigt markiert und ggf. anders dargestellt

**Geschäftsprozess 3: Überfällige Aufgaben anzeigen**

1. Beim Start prüft das Programm alle Aufgaben
2. Aufgaben mit Fälligkeitsdatum < wie aktuelles Datum werden als „überfällig“ markiert/angezeigt

**Geschäftsprozess 4: Aufgabe löschen**

1. In der Aufgabenliste wählt der Nutzer eine Aufgabe aus
2. Mit Klick auf „Aufgabe löschen“ wird die Aufgabe entfernt (komplett gelöscht)

Eine Aktivität beschreibt in UML-Aktivitätsdiagrammen einen Ablauf von Aktionen, die zusammen eine bestimmte Funktionalität oder ein Verhalten implementieren. Dabei kann eine Aktivität sowohl eine einfache Aktion als auch eine komplexe, strukturierte Verhaltensweise darstellen (Balzert, 2009, S. 236). Abbildung 3 zeigt ein UML-Aktivitätsdiagramm für den Kernprozess des Anlegens einer Aufgabe.

Abbildung 3: UML-Aktivitätsdiagramm für Kernprozess "Aufgabe anlegen"



Quelle: Eigene Darstellung mit Microsoft Visio (Microsoft 365, Version 2506)

# Geschäftsregeln

* Eine Aufgabe muss immer mindestens einen Titel besitzen
* Das Fälligkeitsdatum einer Aufgabe darf nicht in der Vergangenheit liegen
* Eine Aufgabe kann erst gespeichert werden, wenn alle Pflichtfelder (Titel) ausgefüllt sind
* Jede Aufgabe ist eindeutig identifizierbar (z.B. durch eine ID)
* Bereits erledigte Aufgaben können nicht bearbeitet werden (nur noch löschen möglich)
* Das Fälligkeitsdatum von bereits erledigten Aufgaben führt nicht mehr zu einem Zustandswechsel (bspw. Verändern der Schrift von grün auf rot o.Ä.)
* Beim Löschen einer Aufgabe öffnet sich ein Modal, um unbeabsichtigtes löschen zu verhindern
* Alle angegebenen Daten werden beim Schließen der Anwendung gespeichert
* Das Fälligkeitsdatum ist optional, aber wenn es gesetzt wird, muss es nach dem Erstellungsdatum liegen

# Systemschnittstellen

In diesem Projekt sind keine externen technischen Schnittstellen (z.B. HTTP, FTP) vorgesehen. Die Anwendung läuft komplett lokal auf dem Rechner des Benutzers. Der Datenformat der Aufgaben wird zum Beispiel TXT, CSV oder JSON sein.

# Benutzerschnittstellen

Die Anwendung ist als klassische Desktop-Fensteranwendung unter Windows mit Qt6 realisiert. Sie besteht aus einem Hauptfenster sowie mehreren Dialogfenstern, die jeweils klar abgegrenzte Aufgabenbereiche abbilden.

## 14.1 Struktur der Oberfläche

* Hauptfenster: Zeigt alle Aufgaben übersichtlich als Liste
* Neue Aufgabe anlegen: Button „+ Aufgabe anlegen“ öffnet Dialog
* Aufgabendetails: Im Dialog kann der Nutzer die Attribute eingeben
* Löschen/Erledigt: Neben jeder Aufgabe befinden sich Buttons für „Löschen“ oder „als erledigt markieren“
* Kalender: Der Kalender soll im Hauptfenster angezeigt werden

## 14.2 Wichtigste Dialoge & Abläufe

**Neue Aufgabe anlegen**

* Klick auf „+ Aufgabe hinzufügen“
* Eingabefeld für Titel (Pflichtfeld)
* Eingabefeld für Fälligkeitsdatum (optional)
* Eingabefeld für Beschreibung (optional)
* „Speichern“ Button, der nur aktiv ist, wenn das Pflichtfeld ausgefüllt ist
* Fehlermeldung, falls Pflichtfeld leer bleibt oder Datum ungültig ist

**Aufgabe löschen**

* Klick auf „Löschen“-Button öffnet ein Modal/Bestätigungsdialog und erst danach wird die Aufgabe gelöscht

**Eingabevalidierung**

* Titel muss ausgefüllt sein
* Fälligkeitsdatum darf nicht in der Vergangenheit liegen

## 14.3 Skizze der Anwendung

Abbildung 4 zeigt eine grobe Skizze der grafischen Benutzeroberfläche.

Abbildung 4: Skizze Aufgabenplaner

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

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung mit Microsoft Visio (Microsoft 365, Version 2506)

# Systemkontext und Datenflüsse

Der Systemkontext beschreibt die relevante Umgebung, die es um ein System herum gibt und die daher in der Entwicklung von Systemen zu beachten ist. Es gibt jedoch auch eine irrelevante Umgebung, die keinen Einfluss auf die Entwicklung hat und mit einer Grauzone an die relevante Umgebung anschließt. Die Interpretation einer Anforderung wird daher durch den Kontext beeinflusst (Balzert, 2009, S. 462), weshalb er ein fester Teil der ersten Phase des Softwareentwurfsprozesses ist (Sommerville, 2012, S. 217).

## 15.1 Anwendungsspezifischer Kontext

Im Fall dieser Anwendung ist das System vollständig lokal auf dem Windows-Rechner installiert. Es handelt sich um eine Desktop-Fensteranwendung, die weder Cloud-Dienste noch Netzwerkschnittstellen verwendet.

Die wichtigsten Quellen (Dateneingabe) sind:

* Der Benutzer, der über die GUI Aufgaben erstellt, bearbeitet oder löscht
* Eine lokale Datei (TXT, JSON oder CSV), die beim Start der Anwendung automatisch eingelesen wird und dann die Aufgaben in das Programm lädt

Die wichtigsten Senken (Daten-Ausgaben) sind:

* Die GUI, welche Aufgaben und Statusänderungen visuell darstellt
* Eine lokale Datei, in die alle Änderungen der aktuellen Sitzung gespeichert werden, sobald der Benutzer eine Aufgaben hinzufügt oder bearbeitet

Externe Systeme wie APIs, Server oder Datenbanken sind in diesem Projekt nicht vorhanden.

## 15.2 Abgrenzung und Kontextdarstellung

Das System interagiert ausschließlich mit dem Betriebssystem (Datei I/O) und dem Benutzer. Es existieren keine weiteren externen Schnittstellen oder Abhängigkeiten in diesem Projekt. Abbildung 5 zeigt die Abgrenzung des Systemkontextes mit Hilfe eines einfachen Kontextdiagrammes.

Abbildung : Kontextdiagramm

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

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung mit Microsoft Visio (Microsoft 365, Version 2506)

# Phase 2: Erarbeitungs- und Reflexionsphase – Architekturdokument

# Technologieübersicht

## 16.1 Programmiersprache

Die für dieses Projekt ausgewählte Programmiersprache ist C++, folgend werden die wichtigsten Gründe für die Auswahl erläutert.

* Motivation: Vertiefung der eigenen Kenntnisse, wichtig für Ingenieurs- und Software-Engineering Berufe
* Fördert allgemeines Verständnis für Speicherverwaltung, effiziente Algorithmen und systemnahe Entwicklung
* Große Ökosystem, viele Libraries und plattformübergreifend einsetzbar
* Auch wenn Entwicklung schwerer als bspw. mit Python ist, ist der Lerneffekt für Studium/Beruf höher
* GUI-Anbindung mit Qt

## 16.2 Frameworks

Die grafische Oberfläche wurde mit dem Qt6-Framework (Qt 6, Version 17.0.0) als klassische Fensteranwendung für Windows realisiert. Es kommen QMainWindow sowie mehrere QDialog-basierte Komponenten zum Einsatz. Auch das googletest Framework wird für die späteren Tests implementiert.

* Qt6
  + Wird für grafische Oberfläche (GUI) verwendet
  + Ermöglicht plattformübergreifende Entwicklung für Fensteranwendungen
  + Gute Unterstützung für UI-Design, Events und eigene Widgets
  + Integration in C++ Projekt und Build-System ist Standard
* Googletest
  + Framework für automatisierte Unit-Tests
  + Weit verbreitet im C++ Umfeld, auch im industriellen Einsatz
  + Moderne C++ Features für Testfälle und Test Suites
  + Bringt professionelle Entwicklungsmethodik (Test Driven Development, Absicherung von Code…)

## 16.3 Bibliotheken

Bisher werden im Projekt folgende Bibliotheken verwendet, diesen können sich jedoch je nach Projektablauf noch verändern.

* C++ Standardbibliothek (STL): Für Vektoren, Strings, Maps usw.
* QtCore, QtGui, QtWidgets

## 16.4 Entwicklungswerkzeuge

* Visual Studio Code (Windows, 2021, Version 1.102.3)
* Qt Creator (Qt 6, Version 17.0.0)
* CMake als Buildsystem (CMAKE, Version 4.0.0-rc3)
* GitHub zur Versionskontrolle (GitHub, Version 2.46.0.windows.1)
* windeployqt zum Erstellen von Release-Paketen (Qt For Windows - Deployment | Version 17.0.0)

# Architekturübersicht

In diesem Projekt wird wie bereits vorher genannt das MVC-Muster (Model-View-Controller) umgesetzt. Es ermöglicht eine saubere Trennung von Benutzerschnittstelle, Anwendungslogik und Datenmodell, wodurch die Software leichter erweiterbar und testbar ist (Ludin, 2014, S. 97). Um eine lose Kopplung zu erreichen, wurde außerdem ein Interface implementiert.

Der Controller (TaskController) hält einen Zeiger auf das Interface (TaskInterface) und ruft Methoden wie addTask(), deleteTask(), loadAufgaben() und saveAufgaben() ausschließlich über das diese Interface auf. Das Model (TaskManager) implementiert das Interface und stellt die Logik für die Aufgabenverwaltung bereit. Die View (mainwindow) kommuniziert ausschließlich mit dem Controller und kennt damit weder die konkrete Implementierung des Models noch die Datenstrukturen.

* **View (Benutzerschnittstelle):**
  + Besteht aus den MainWindow und den Dialogklassen
  + Verantwortlich für Darstellung der Daten und Erfassung von Benutzereingaben
  + Kommuniziert nur mit dem Controller
* **Controller (TaskController):**
  + Vermittelt zwischen View und Model
  + Hält einen Zeiger auf das Interface
  + Ruft Methoden des Models ausschließlich über das Interface auf
* **Interface (TaskInterface):**
  + Definiert abstrakt die vom Model bereitgestellten Methoden
  + Ermöglicht lose Kopplung zwischen Controller und Model
  + Controller nutzt das Interface, Model implementiert es
* **Modell (Datenhaltung & Logik):**
  + Umfasst Task, TaskManager sowie die Dateioperationen (CSV)
  + Implementiert das Interface und enthält die eigentliche Logik
  + Verantwortlich für das Speichern, Laden und Verwalten der Aufgabenobjekte

# Interfaces

Ein Interface beschreibt eine abstrakte Schnittstelle, die konkrete Klassen implementieren müssen. Andere Teile des Systems kommunizieren ausschließlich über dieses Interface, kennen also nur die definierten Methoden und nicht die konkrete Umsetzung (Balzert, 2009, S. 40–41). Dadurch wird sichergestellt, dass einzelne Module problemlos ausgetauscht oder erweitert werden können, ohne dass Anpassungen in anderen Komponenten notwendig sind.

In dieser Anwendung werden drei zentrale Interfaces verwendet:

* TaskInterface: Das Interface für das Model. Es legt alle Methoden zur Verwaltung der Aufgaben fest und wird vom TaskManager implementiert.
* TaskControllerInterface: Das Interface für den Controller. Es definiert die vom Controller bereitzustellenden Funktionen, z.B. das Hinzufügen oder Löschen von Aufgaben. Die Klasse TaskController implementiert das Interface.
* ViewInterface: Das Interface für die View. Es gibt vor, wie Aufgaben und Fehlermeldungen angezeigt werden sollen. Die Klasse MainWindow setzt dieses Interface um.

Die Verwendung von Interfaces hat folgende Vorteile:

* Austauschbarkeit: Unterschiedliche Implementierungen für die View können einfach bereitgestellt werden (z.B. GUI, Konsole, Web) und sind zukunftssicher (Balzert, 2009, S. 42).
* Erweiterbarkeit: Neue Anforderungen oder alternative Bedienkonzepte (z.B. weitere Views) lassen sich so einfach integrieren, ohne das restliche System anzupassen (Balzert, 2009, S. 42).
* Wartbarkeit: Der Code bleibt übersichtlich, da Änderungen an einer Schicht nicht zwangsläufig Auswirkungen auf andere Schichten haben (Balzert, 2009, S. 42).

# Struktur

Im folgenden Kapitel wird die Struktur der Anwendung aufgezeigt. Abbildung 6 zeigt anhand eines UML-Klassendiagrammes, welche Klassen, Attribute und Methoden in der Anwendung verwendet werden. Um eine bessere Übersichtlichkeit zu gewährleisten, werden nur für das Verständnis relevante Methoden und Attribute aufgezeigt.

## 19.1 Hauptkomponenten

* **MainWindow (View)**
  + Verantwortlich für das Hauptfenster und die Darstellung der Aufgabenliste
  + Nimmt Benutzereingaben entgegen (z.B. Buttons, Doppelklicks)
  + Kommuniziert ausschließlich mit dem Controller
  + Implementiert das Interface ViewInterface
* **DialogTaskHinzufuegen / DialogTaskBearbeiten (View)**
  + Separate Dialogfenster für das Hinzufügen bzw. Bearbeiten einer Aufgabe
  + Erfassen und Bearbeiten der Aufgabendaten durch den Benutzer
  + Ergebnisse werden an MainWindow (und damit Controller) weitergereicht
* **erledigteAufgabenliste / ueberfaelligeAufgaben (View)**
  + Dialogfenster zur Anzeige erledigter bzw. überfälliger Aufgaben
* **TaskController (Controller)**
  + Vermittelt zwischen View und Model über das Interface
  + Nimmt Aufrufe der View entgegen, ruft Methoden am Interface (Model) auf
  + Kapselt Steuerungslogik (z.B. Validierung, Übergabe an Model, Aufruf von Speichern/Laden
  + TaskController nutzt die TaskFactory, um neue Aufgaben anzulegen.
  + Implementiert das Interface TaskControllerInterface
* **TaskManager (Model)**
  + Implementiert das Interface, verwaltet die Sammlung aller Aufgaben
  + Verantwortlich für das Speichern, Laden, Hinzufügen, Löschen, Bearbeiten, Suchen der Aufgaben
  + Arbeitet ausschließlich über das Interface mit dem Controller
  + Die Erzeugung neuer Task-Objekte erfolgt zentral über die Klasse TaskFactory (nach Factory-Pattern)
  + Implementiert das Interface TaskInterface
* **Task**
  + Datenklasse für einzelne Aufgaben (Attribute: Titel, Beschreibung, Fälligkeitsdatum, Status, Index)
  + Methoden zum Ändern und Auslesen der Aufgabendaten

## 19.2 Abhängigkeiten

* TaskController und TaskManager verwenden die TaskFactory zur Erzeugung von Task-Objekten (Factory-Pattern)
* MainWindow und die Dialoge kommunizieren ausschließlich mit dem TaskController
* TaskController hält einen Zeiger auf das Interface (TaskInterface) und ruft Methoden wie addTask(), deleteTask(), loadAufgaben(), saveAufgaben() usw. ausschließlich über dieses Interface auf
* TaskManager implementiert das Interface und stellt die Logik zur Aufgabenverwaltung bereit
* TaskManager verwaltet eine Sammlung von Task-Objekten
* Dialogfenster übergeben ihre Eingaben über MainWindow an den Controller
* Die View kennt niemals die konkrete Implementierung des Models, sondern nutzt immer den Controller

Abbildung : UML-Klassendiagramm



Quelle: Eigene Darstellung mit Microsoft Visio (Microsoft 365, Version 2506)

## 19.3 Erweiterbarkeit

Die lose Kopplung durch die Interfaces ermöglicht es, Module auszutauschen (z. B. durch eine neue Klasse für Datenbank- oder Netzwerkspeicherung), ohne dass aufwendige Änderungen an der Code Basis notwendig sind. Neue Features wie alternative Speicherformate, Synchronisierung oder weitere Darstellungsformen können einfach durch neue Implementierungen des Interfaces ergänzt werden. Das MVC-Muster vereinfacht das Testen einzelner Komponenten, da View, Controller und Model isoliert getestet werden können.

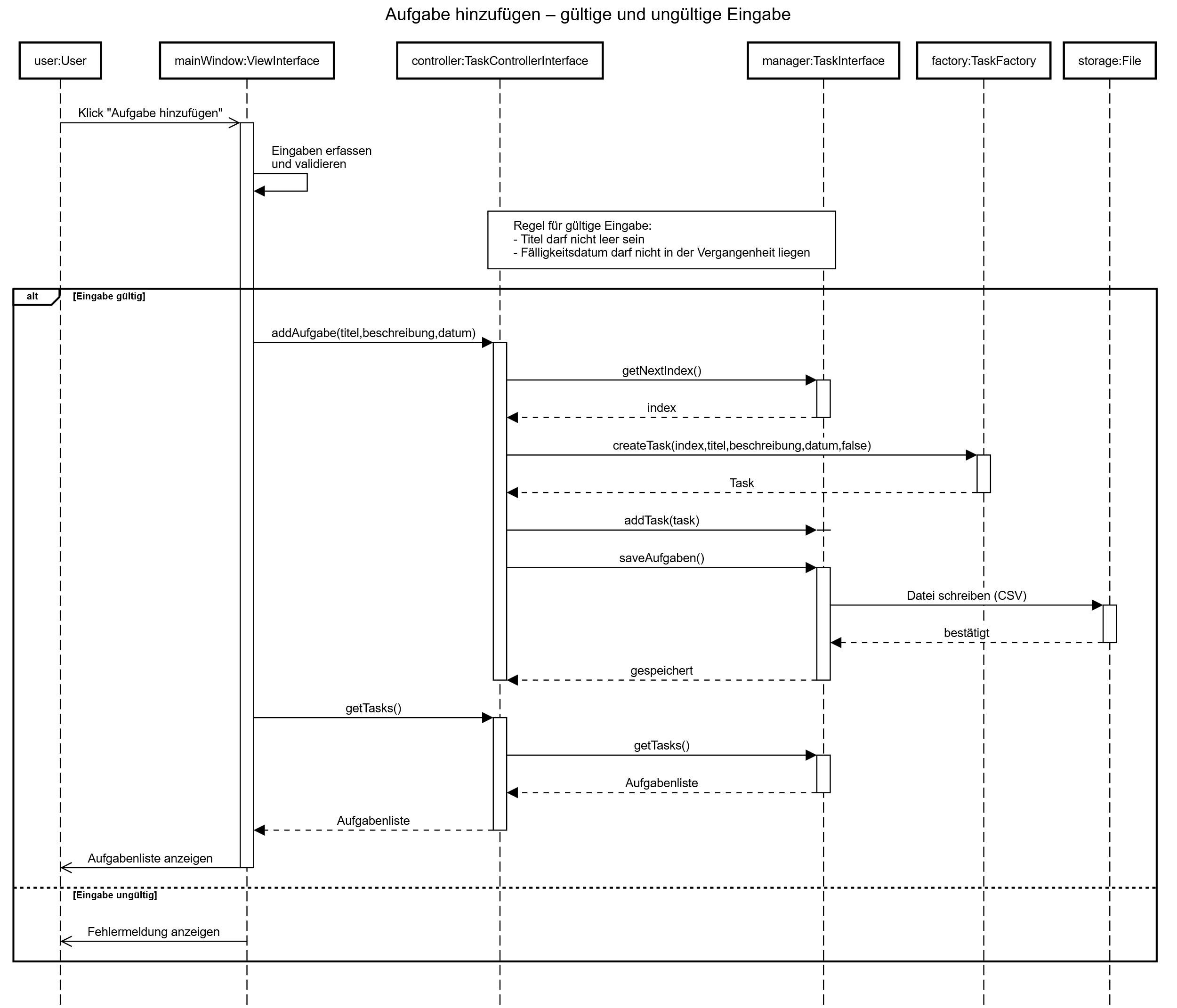
# Verhalten

In der UML werden Sequenzdiagramme eingesetzt, um die Kommunikation und den Ablauf zwischen verschiedenen Objekten bzw. Komponenten einer Anwendung darzustellen. Sequenzdiagramme können mit Anwendungsfällen verknüpft werden und bieten eine Sicht auf den Ablauf der Methodenaufrufe und Antworten zwischen den beteiligten Instanzen (Sommerville, 2012, S. 162). Die Objekte und Akteure werden als Rechtecke mit Lebenslinien visualisiert, die Interaktionen als gerichtete Kanten (Pfeile) für Methodenaufrufe und Rückgaben (Balzert, 2009, S. 333–334). Zudem wird im Diagramm durch einen ALT-Block auch der Fehlerfall (z. B. ungültige Eingaben) dargestellt, wie es in Sommerville (2012, S. 162) empfohlen wird. Die Rückmeldung an den Benutzer erfolgt dann direkt über die grafische Oberfläche, ohne weitere Systeminteraktionen.

Das folgende Sequenzdiagramm in Abbildung 7 zeigt den Ablauf des Use Cases „Aufgabe anlegen“ (sh. Kapitel 11 – Geschäftsprozess 1). Im Diagramm ist zu erkennen, dass die Benutzeraktion zunächst an die View weitergegeben wird, die wiederum den Controller aufruft. Über das Interface wird das Model (TaskManager) angesprochen, was die Daten an die Persistenzschicht weiterleitet. Nach Abschluss der Operation werden Rückgabepfade von TaskManager über das Interface zum Controller und von dort zur View sowie als Rückmeldung an den Benutzer dargestellt.

Die Darstellung als Sequenzfluss wurde gewählt, um den logischen Ablauf und die zeitliche Reihenfolge der wichtigsten Interaktionen im Use Case „Aufgabe anlegen“ klar und übersichtlich zu veranschaulichen. Damit wird nachvollziehbar, wie die Benutzeraktion Schritt für Schritt durch alle beteiligten Komponenten des Systems weitergegeben und verarbeitet wird.

Abbildung 7: UML-Sequenzdiagramm: Aufgabe hinzufügen



Quelle: Eigene Darstellung mit (SequenceDiagram.org, o. D.)

# Phase 3: Finalisierungsphase – Testdokument

# Teststrategie

## 21.1 Unit-Tests

Zum Testen werden automatisierte Unit-Tests mit Googletest geschrieben, wie in Abschnitt 13.2 bereits erwähnt. Es werden die Kernfunktionen der Geschäftslogik (Model) durch die automatisierten Tests abgesichert. Konkret werden folgende Funktionen bzw. Module getestet:

* Prüfen ob TaskFactory::createTask alle Attribute korrekt in ein neues Task-Objekt übernimmt (T01)
* Testet die Kernfunktion addTask des TaskManagers und stellt sicher, dass das Hinzufügen einer Aufgabe die Liste korrekt erweitert (T02)
* Testet die Kernfunktion deleteTask des TaskManagers und stellt sicher, dass eine vorhandene Aufgabe korrekt entfernt wird (T03)
* Testet die Bearbeitungsfunktion editAufgabe des TaskManagers und stellt sicher, dass Änderungen an einer vorhandenen Aufgabe korrekt übernommen werden (T04)
* Testet die Funktion markiereAufgabeAlsErledigt des TaskManagers und stellt sicher, dass der Status einer Aufgabe von false auf true wechselt (T05)
* Testet die Persistenzfunktionalität des TaskManagers. Es wird geprüft, ob die Methoden saveAufgaben und loadAufgaben korrekt in eine CSV-Datei schreiben und anschließen wieder einlesen

Ziel der Unit-Tests ist es, sicherzustellen, dass die wichtigsten Kernfunktionen bzw. Module der Anwendung korrekt funktionieren.

## 21.2 Integrationstests

Zusätzlich wurden Integrationstests durchgeführt, bei denen ein typischer Workflow aus Anwendersicht getestet wurde. Dabei wurde automatisiert getestet:

* Aufgabe anlegen (T05)
* Angelegte Aufgabe bearbeiten (T05)

Außerdem wurde ergänzend getestet, ob nach dem Markieren einer Aufgabe als erledigt diese wie vorhergesehen aus dem ListWidget der offenen Aufgaben verschwindet (T05). Diese Funktion lässt sich nur manuell in der GUI überprüfen und wird daher mit einem Screenshot im Testprotokoll hinterlegt.

Das Ziel der Integrationstests ist es, sicherzustellen, dass das Zusammenspiel der Anwendungsabläufe vom Benutzer wie erwartet funktionieren.

## 21.3 Systemtests

Die Systemtests wurden komplett manuell durchgeführt. Dabei wurde die komplette Anwendung inklusive grafischer Benutzeroberfläche getestet. Die Testergebnisse werden zur Nachvollziehbarkeit durch Screenshots dokumentiert und dem Testprotokoll beigefügt. Folgende Funktionen wurden geprüft:

* Aufgabe anlegen und bearbeiten über GUI (S01, S02)
* Aufgabe als erledigt markieren (S03)
* Überfällige Aufgaben anzeigen (S04)
* Speichern und Laden der Aufgaben beim Schließen und Öffnen der Anwendung (S05)
* Aufgabe löschen über GUI (S06)

Ziel der Systemtests ist es, die Anwendung unter realen Einsatzbedingungen (nicht aus dem Testumfeld heraus) zu prüfen und sicherzustellen, dass sie aus Anwendersicht stabil und bedienbar ist.

# Testprotokoll

Tabelle 2: Unit-Tests

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test-ID** | **Kurzbeschreibung** | **Vorbedingung** | **Aktion** | **Erwartetes Ergebnis** | **Tatsächliches Ergebnis** |
| T01 | Prüft ob TaskFactory::createTask ein Task-Objekt mit Parametern erzeugt | TaskFactory ist implementiert | Aufruf von TaskFactory::createTask(1, "Testtitel", "Testbeschreibung", "2025-12-31", false) | Das zurückgegebene Objekt besitzt: Index = 1, Titel = „Testtitel“, Beschreibung = „Testbeschreibung“, Faelligkeitsdatum = „2025-12-31“ und IstErledigt = false | Wie erwartet |
| T02 | Hinzufügen vergrößert die Liste | Leerer TaskManager | addTask(Task(…)) aufrufen | Aufgabenzahl = vorher + 1 | Wie erwartet |
| T03 | Entfernen verkleinert die Liste | TaskManager mit 2 Aufgaben | deleteTask(0) | Aufgabenzahl - 1 | Wie erwartet |
| T04 | Titel, Beschreibung und Datum ändern | TaskManager mit einer Aufgabe | editAufgabe(0, „Neu“, „NeueDesc“, „2025-12-31“) | Aufgabe hat neue Werte, Größe gleich | Wie erwartet |
| T05 | Aufgabe erledigt markieren | TaskManager mit einer Aufgabe (istErledigt = false) | markiereAufgabeAlsErledigt(0) | Status der Aufgabe muss = true sein | Wie erwartet |
| T06 | Persistenz (Speichern und Laden) | TaskManager hat 2 Aufgaben | saveAufgaben() aufrufen; neuen Manager erzeugen, loadAufgaben() ausführen | Beide Aufgaben werden aus CSV geladen und besitzen dieselben Attribute wie vor dem speichern | Wie erwartet |

Quelle: Eigene Darstellung

Tabelle 1: Integrationstest

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test-ID** | **Kurzbeschreibung** | **Vorbedingung** | **Aktion** | **Erwartetes Ergebnis** | **Tatsächliches Ergebnis** |
| I01 | Typischer Anwender-Workflow: Aufgabe anlegen und bearbeiten; Aufgabe als erledigt markieren | Leerer TaskManagere | addAufgabe(…), danach editAufgabe(…) mit anderen Werten; Aufgabe als erledigt markieren | Aufgabe hat neue Werte (nach edit) und Aufgabenanzahl bleibt 1; Aufgabe verschwindet in Liste der erledigten Aufgaben | Wie erwartet |

Quelle: Eigene Darstellung

**Zusatz zu Integrationstest I01**

* Schritt 1 (Vorbereitung): Aufgabe anlegen
* Schritt 2: ausgewählte Aufgabe als erledigt markieren
* Schritt 3: Prüfen, ob die Aufgabe in der Liste der erledigten Aufgaben ist

Abbildung 8: Integrationstest I01

Ein Bild, das Text, Screenshot, Software, Webseite enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

**Systemtests**

Tabelle 2: Tabelle Systemtests

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test-ID** | **Kurzbeschreibung** | **Vorbedingung** | **Aktion** | **Erwartetes Ergebnis** | **Tatsächliches**  **Ergebnis** |
| S01 | Aufgabe anlegen (GUI) | Anwendung geöffnet | Über GUI Aufgabe anlegen | Aufgabe erscheint in Liste | Wie erwartet |
| S02 | Aufgabe bearbeiten (GUI) | Aufgabe vorhanden | Aufgabe per GUI bearbeiten | Änderungen werden übernommen | Wie erwartet |
| S03 | Aufgabe als erledigt markieren (GUI) | Aufgabe vorhanden | Aufgabe per GUI als erledigt markieren | Aufgabe verschwindet aus offener Liste | Wie erwartet |
| S04 | Überfällige Aufgaben anzeigen | Überfällige Aufgabe vorhanden | Überfällige Aufgabe über GUI anzeigen | Überfällige Aufgabe wird korrekt markiert und in Liste angezeigt | Wie erwartet |
| S05 | Speichern und Laden testen | Aufgabe vorhanden | Anwendung schließen und öffnen | Aufgaben werden korrekt geladen | Wie erwartet |

Quelle: Eigene Darstellung

Abbildung 9: Systemtest S01 - Aufgabe anlegen

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

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Abbildung : Systemtests S02 - Aufgabe bearbeiten

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

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Abbildung : Systemtests S03 - Aufgabe erledigt

Ein Bild, das Text, Screenshot, Software, Webseite enthält.

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Abbildung : Systemtests S04 - Aufgabe überfällig

***Ein Bild, das Text, Screenshot, Software, Webseite enthält.

KI-generierte Inhalte können fehlerhaft sein.***

Quelle: Eigene Darstellung

Abbildung : Systemtests S05 - Speichern/Laden

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

KI-generierte Inhalte können fehlerhaft sein.

Quelle: Eigene Darstellung

Abbildung : Systemtests S06 - Aufgabe löschen/entfernen

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

KI-generierte Inhalte können fehlerhaft sein.**

Quelle: Eigene Darstellung

# Abstract

Im vorliegenden Portfolioprojekt wird eine Aufgabenplaner-Anwendung für Windows entwickelt. Die Anwendung wurde mit C++ und dem Qt Framework erstellt und ermöglicht es Privatpersonen, Berufstätigen und Studenten ihren Tag einfach zu planen. Typische Anforderungen an den Aufgabenplaner sind das Anlegen, Bearbeiten, Löschen und Filtern von Aufgaben. Die Oberfläche wurde benutzerfreundlich gestaltet und dient gleichzeitig als praktisches Beispiel für die Anwendung objektorientierter Programmierung. Im Fokus stand nicht nur die Anwendung selbst, sondern auch der gesamte Entwicklungsprozess inklusive der Anforderungsdefinition, Testen und Deployment.

Die entwickelte Architektur basiert auf dem MVC-Muster. Das Model (TaskManager) verwaltet alle Aufgaben, kapselt die Geschäftslogik und speichert die Daten dauerhaft in einer CSV-Datei (Persistenz). Die View (Dialog- und Hauptfenster) stellt die grafische Benutzeroberfläche bereit. Für die Erzeugung neuer Aufgaben kommt eine Factory-Klasse (TaskFactory) zum Einsatz, um eine lose Kopplung zu gewährleisten.

Zu Beginn der Projektarbeit (Phase 1) stand die Konzeption an. Potenzielle Risiken wurden mit Frühwarnindikatoren identifiziert und Gegenmaßnahmen definiert. Außerdem wurde eine Zeitplanung mittels Gantt-Diagramm erstellt, welche vor allem wegen fehlender Puffer deutlich zu knapp geplant wurde. Danach wurde eine Zielgruppenanalyse durchgeführt und funktionale und nicht-funktionale Anforderungen festgelegt. Auf dieser Grundlage fiel die Entscheidung für eine klassische Desktop-Applikation mit C++ und Qt, um Oberfläche und Logik sauber zu trennen.

Im weiteren Verlauf lag der Schwerpunkt auf der Umsetzung des Designs. Klassen wie Task und TaskManager wurden entwickelt, um Aufgaben als Objekte abzubilden und innerhalb der Anwendung verwalten zu können. Für die verschiedenen Dialogfenster (z. B. Hinzufügen und Entfernen) wurden separate Klassen mit spezifischer Logik erstellt. Die Programmstruktur könnte künftig noch stärker von Vererbungen profitieren (etwa durch eine Basisklasse für Dialoge), diese Erkenntnis wird für kommende Projekte berücksichtigt.

Der Umgang mit Dateien und Pfadangaben sowie die Ordnerstruktur stellte sich als zentrales Thema heraus: Die Nutzung eines OneDrive‑Ordners führte zu Build‑Problemen, die erst nach dem Wechsel auf einen lokalen Ordner und der Umstellung von absoluten auf relative Include-Pfadangaben behoben werden konnten. Für zukünftige Projekte empfiehlt es sich, frühzeitig eine klare Ordnerstruktur festzulegen und die Einbindung externer Frameworks wie GoogleTest oder Qt systematisch auf jeder Ebene zu planen.

In Phase 3 wurde der Schwerpunkt auf die Qualitätssicherung gelegt. Es wurden automatisierte Unit-Tests entwickelt, welche die Kernfunktionen des Models abdecken. Die Erzeugung von Aufgaben über die Factory, das Hinzufügen, Bearbeiten und Löschen von Aufgaben, das Setzen des Erledigt-Status sowie das Speichern und Laden der Daten wurden als Unit-Tests getestet.

Ein Integrationstest prüft den typischen Workflow (Anlegen, Bearbeiten, Erledigen), während Systemtests die GUI-Abläufe dokumentieren, die sich nicht automatisieren lassen. Ein weiterer Knackpunkt war das Deployment. Damit Qt‑Anwendungen auf anderen Rechnern lauffähig sind, müssen die benötigten DLLs eingebunden werden. Anfangs führte das zu Fehlermeldungen, mit dem Tool windeployqt ließ sich das Deployment jedoch automatisieren, indem alle erforderlichen Bibliotheken in den Release-Ordner kopiert wurden.

Die Verzögerungen im Zeitplan traten vor allem durch wiederholte Build-Probleme oder den Wechsel zwischen den Compilern MinGW und MSVC auf, blieben jedoch ohne gravierende Auswirkungen. Das Projektziel eine stabile, getestete Aufgabenplaner-Anwendung mit klarer MVC-Struktur, lokaler Persistenz und benutzerfreundlicher Oberfläche zu entwickeln wurde erreicht. Das Projekt bot wertvolle Erfahrungen im Umgang mit GitHub, Architekturentscheidungen, Syntax, Testautomatisierung und Deployment. Für kommende Vorhaben empfiehlt sich, von Beginn an eine klare OOP-Struktur, ein umfassendes Testdesign und eine sorgfältige Projektorganisation zu etablieren; die Portfoliodokumentation zeigte, wie hilfreich Reflexion und Dokumentation sind, um Schwachstellen zu erkennen und konkrete Verbesserungsansätze zu erkennen.

# Quellenverzeichnis

Balzert, H. (2009). Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering. In Spektrum Akademischer Verlag eBooks. https://doi.org/10.1007/978-3-8274-2247-7

Boch, J. (2018). *Effective java: Best Practices für die Java-Plattform*.

CMAKE - Upgrade your software build system. (o. D.). Version 4.0.0-rc3 https://cmake.org/ Lizenz: BSD 3-Clause License (https://cmake.org/licensing/)

Download Visual Studio Code - Mac, Linux, Windows. (2021, 3. November). Version 1.102.3 https://code.visualstudio.com/download Lizenz: MIT License (https://code.visualstudio.com/license)

GitHub Build and ship software on a single, collaborative platform. (2025). Version 2.46.0.windows.1 https://github.com/ Lizenz: GitHub Desktop/Git: GPLv2 (Git), MIT License (GitHub Desktop)

google. (o. D.). Version 1.17.0 GitHub - google/googletest: GoogleTest - Google Testing and Mocking Framework. GitHub. https://github.com/google/googletest Lizenz: BSD-3-Clause (https://github.com/google/googletest/blob/main/LICENSE)

Ludin, A. (2014). Learn BlackBerry 10 App Development: a Cascades-Driven approach. https://directory.doabooks.org/handle/20.500.12854/51477

Microsoft Excel: Kostenlose Online-Tabellenkalkulationssoftware | Microsoft 365. (o. D.). Version 2506 https://www.microsoft.com/de-de/microsoft-365/excel?market=de Lizenz: Proprietär, kommerziell

Microsoft Visio: Diagramming & Flowcharts | Microsoft 365. (o. D.). Version 2506 https://www.microsoft.com/en-us/microsoft-365/visio/flowchart-software Lizenz: Proprietär, kommerziell

Qt 6 - The latest version of Qt. (o. D.). Version 17.0.0 https://www.qt.io/product/qt6 Lizenz: GPL/LGPL (freie Version) oder kommerziell erhältlich (siehe https://www.qt.io/licensing/)

Rau, K. & Schuster, T. (2021). Agile objektorientierte Software-Entwicklung: Schritt für Schritt vom Geschäftsprozess zum Java-Programm. Springer Vieweg.

SequenceDiagram.org - UML Sequence Diagram online tool. (o. D.). https://sequencediagram.org/

Silberbauer, C. (2020). *Einstieg in Java und OOP: Grundelemente, Objektorientierung, Design-Patterns und Aspektorientierung*. Springer Vieweg.

Sommerville, I. (2012). Software Engineering. ISBN: 978-3-86894-099-2

Windeployqt Qt for Windows - Deployment | Qt 6.9. (o. D.). Version 17.0.0 https://doc.qt.io/qt-6/windows-deployment.html Lizenz: Lizenz: GPL/LGPL oder kommerziell

# Anhang

**Quellcode**

Datei „CMakeLists.txt“

set(CMAKE\_AUTOUIC ON)

set(CMAKE\_AUTOMOC ON)

set(CMAKE\_AUTORCC ON)

cmake\_minimum\_required(VERSION 3.16)

project(AufgabenTracker LANGUAGES CXX)

if (MSVC)

add\_compile\_options(/std:c++17 /Zc:\_\_cplusplus)

endif()

set(CMAKE\_CXX\_STANDARD 17)

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

set(CMAKE\_PREFIX\_PATH "C:/Qt/6.6.2/mingw\_64/lib/cmake")

find\_package(Qt6 COMPONENTS Widgets REQUIRED)

add\_executable(AufgabenTracker

main.cpp

Task.cpp

Task.h

TaskManager.cpp

TaskManager.h

TaskController.h

TaskController.cpp

TaskFactory.h

TaskFactory.cpp

../Aufgabenplaner/mainwindow.cpp

../Aufgabenplaner/mainwindow.h

../Aufgabenplaner/mainwindow.ui

../Aufgabenplaner/dialogtaskbearbeiten.cpp

../Aufgabenplaner/dialogtaskbearbeiten.h

../Aufgabenplaner/dialogtaskbearbeiten.ui

../Aufgabenplaner/dialogtaskhinzufuegen.cpp

../Aufgabenplaner/dialogtaskhinzufuegen.h

../Aufgabenplaner/dialogtaskhinzufuegen.ui

../Aufgabenplaner/erledigteaufgabenliste.cpp

../Aufgabenplaner/erledigteaufgabenliste.h

../Aufgabenplaner/erledigteaufgabenliste.ui

../Aufgabenplaner/ueberfaelligeaufgaben.cpp

../Aufgabenplaner/ueberfaelligeaufgaben.h

../Aufgabenplaner/ueberfaelligeaufgaben.ui

)

target\_link\_libraries(AufgabenTracker Qt6::Widgets)

# Test-Setup

add\_subdirectory(googletest)

add\_executable(simple\_test

test\_simple.cpp

Task.cpp

TaskManager.cpp

TaskController.cpp

TaskFactory.cpp)

target\_link\_libraries(simple\_test PRIVATE Qt6::Core gtest gtest\_main)

enable\_testing()

add\_test(NAME simple\_test COMMAND simple\_test)

Datei „Task.h“

#include <iostream>

#pragma once

using namespace std;

class Task

{

public:

    Task(int index, const string& titel, const string& beschreibung, const string& faelligkeitsdatum, bool isterledigt);    //Standard Konstruktor

    ~Task();                                                                                                                //Destruktor

    //getter Methoden

    int getIndex() const;

    string getTitel() const;

    string getBeschreibung() const;

    string getFaelligkeitsdatum() const;

    bool getIstErledigt() const;

    //setter Methoden

    void setIndex(const int);

    void setTitel(const string&);

    void setBeschreibung(const string&);

    void setFaelligkeitsdatum(const string&);

    void setIstErledigt(const bool);

private:

    //Attribute

    int index;

    string titel;

    string beschreibung;

    string faelligkeitsdatum;

    bool istErledigt;

};

Datei „Task.cpp“

#include "Task.h"

//Konstruktor

Task::Task(int i, const string& t, const string& b, const string& f, bool e)

: index(i), titel(t), beschreibung(b), faelligkeitsdatum(f), istErledigt(e)

{

                                                                            //Attribute bereits in Intialisierungsliste gesetzt

}

int Task::getIndex() const                                                  //Definieren der entsprechenden getter/setter Methoden

{

    return index;

};

string Task::getTitel() const

{

    return titel;};

string Task::getBeschreibung() const

{

    return beschreibung;};

string Task::getFaelligkeitsdatum() const

{

    return faelligkeitsdatum;

};

bool Task::getIstErledigt() const

{

    return istErledigt;

};

void Task::setIndex(const int neuerIndex)

{

    index = neuerIndex;};

void Task::setTitel(const string& neuerTitel)

{

    titel = neuerTitel;};

void Task::setBeschreibung(const string& neueBeschreibung)

{

    beschreibung = neueBeschreibung;};

void Task::setFaelligkeitsdatum(const string& neuesFaelligkeitsdatum)

{

    faelligkeitsdatum = neuesFaelligkeitsdatum;

};

void Task::setIstErledigt(const bool neuesIstErledigt)

{

    istErledigt = neuesIstErledigt;

};

Task::~Task() {}                                                               //Destruktor

Datei „TaskController.h“

#ifndef TASKCONTROLLER\_H

#define TASKCONTROLELR\_H

#include <vector>

#include <string>

#include "Task.h"

#include "TaskInterface.h"

#include "TaskControllerInterface.h"

class TaskController : public TaskControllerInterface

{

    private:

        TaskInterface\* taskInterface;

    public:                                                             //Methoden für die View

        TaskController(TaskInterface\* interface);

        void addAufgabe(const string& titel, const string& beschreibung, const string& faelligkeitsdatum);

        void delAufgabe(int index);

        void editAufgabe(int index, const std::string& neuerTitel, const std::string& neueBeschreibung, const std::string& neuesFaelligkeitsdatum);

        void markiereAufgabeAlsErledigt(int index);

        vector<Task>& getTasks();

        void save();

        void load();

};

#endif

Datei „TaskController.cpp“

#include "TaskController.h"

#include "TaskFactory.h"

TaskController::TaskController(TaskInterface\* interface) : taskInterface(interface){}                             //Methoden für View

void TaskController::addAufgabe(const string& titel, const string& beschreibung, const string& faelligkeitsdatum)

{

    int neuerIndex = taskInterface->getNextIndex();

    Task neueAufgabe = TaskFactory::createTask(neuerIndex, titel, beschreibung, faelligkeitsdatum, false);         //neues Task Objekt erstellen mit Factory

    taskInterface->addTask(neueAufgabe);                                                                          //Objekt in vector pushen

    taskInterface->saveAufgaben();                                                                                //Speichert direkt die Aufgaben nach jeder neuen

}

void TaskController::delAufgabe(int index) {

    taskInterface->deleteTask(index);                                                                             //Erase des Objekts im Vector an der Stelle index

    taskInterface->saveAufgaben();

}

void TaskController::markiereAufgabeAlsErledigt(int index) {

    taskInterface->markiereAufgabeAlsErledigt(index);                                                             //setzt automatisch Objekt an index auf true (istErledigt)

    taskInterface->saveAufgaben();

}

std::vector<Task>& TaskController::getTasks() {

    return taskInterface->getTasks();                                                                             //holt alle aktuellen Aufgaben (aus TaskManager)

}

void TaskController::save() {

    taskInterface->saveAufgaben();                                                                                //save Methode für View

}

void TaskController::load() {

    taskInterface->loadAufgaben();                                                                                //load Methode für View

}

void TaskController::editAufgabe(int index, const std::string& neuerTitel, const std::string& neueBeschreibung, const std::string& neuesFaelligkeitsdatum)

{

    taskInterface->editAufgabe(index, neuerTitel, neueBeschreibung, neuesFaelligkeitsdatum);                      //ermöglicht Eingabe neuer Titel, Beschreibung und Datum

    taskInterface->saveAufgaben();

}

Datei „TaskControllerInterface.h“

#pragma once

#include <string>

#include <vector>

#include "Task.h"

class TaskControllerInterface {

public:

    virtual void addAufgabe(const std::string& titel, const std::string& beschreibung, const std::string& faelligkeitsdatum) = 0;

    virtual void delAufgabe(int index) = 0;

    virtual void editAufgabe(int index, const std::string& neuerTitel, const std::string& neueBeschreibung, const std::string& neuesFaelligkeitsdatum) = 0;

    virtual void markiereAufgabeAlsErledigt(int index) = 0;

    virtual std::vector<Task>& getTasks() = 0;

    virtual void save() = 0;

    virtual void load() = 0;

    virtual ~TaskControllerInterface() = default;

};

Datei “TaskFactory.h”

#pragma once

#include "Task.h"

#include <string>

class TaskFactory

{

public:

    static Task createTask(int index, const std::string& titel, const std::string& beschreibung, //Erstellt ein Task-Objekt mit allen Attributen

                           const std::string& faelligkeitsdatum, bool istErledigt);

};

Datei „TaskFactory.cpp“

#include "TaskFactory.h"

Task TaskFactory::createTask(int index, const std::string& titel, const std::string& beschreibung, //Erstellt ein Task-Objekt mit allen Attributen

                             const std::string& faelligkeitsdatum, bool istErledigt) {

    return Task(index, titel, beschreibung, faelligkeitsdatum, istErledigt);

}

Datei “TaskInterface.h”

#ifndef TASKINTERFACE\_H

#define TASKINTERFACE\_H

#include "Task.h"

class TaskInterface                         //abstrakte Klasse, damit nicht überschrieben werden kann bzw. keine Instanz erstellt werden kann

{

    public:

        virtual void saveAufgaben() = 0;    //pure virtual function

        virtual void loadAufgaben() = 0;

        virtual void addTask(const Task& task) = 0;

        virtual void deleteTask(int index) = 0;

        virtual void editAufgabe(int index, const string& neuerTitel, const string& neueBeschreibung, const string& faelligkeitsdatum) = 0;

        virtual void markiereAufgabeAlsErledigt(int index) = 0;

        virtual int getNextIndex() = 0;

        virtual vector<Task>& getTasks() = 0;

        virtual ~TaskInterface() = default;

};

#endif

Datei “TaskManager.h”

#include <iostream>

#include <vector>

#include "Task.h"

#include "TaskInterface.h"

#pragma once

using namespace std;

class TaskManager : public TaskInterface                               //TaskManager erbt aus der abstrakten Basisklasse

{

public:

    TaskManager();                                                      //Konstruktor

                                                                        //Vorgeschriebenen Methoden aus Interface werden implementiert

    void addTask(const Task& task) override;

    void deleteTask(int index) override;

    void editAufgabe(int index, const string& neuerTitel, const string& neueBeschreibung, const std::string& neuesFaelligkeitsdatum) override;

    void markiereAufgabeAlsErledigt(int index) override;

    const vector<Task>& getTasks() const;                               //Referenz auf alle Aufgaben für die Anzeige in der GUI

    vector<Task>& getTasks();                                           //nicht const getTasks Funktion z.B. zum ändern von Aufgaben

    void saveAufgaben() override;                                       //Speichern mit TaskInterface

    void loadAufgaben() override;                                       //Laden mit Taskinterface

    int getNextIndex() override;

private:

    vector<Task> v\_tasks;

};

Datei „TaskManager.cpp“

#include "TaskManager.h"

#include "TaskFactory.h"

#include <algorithm>

#include <iostream>

#include <fstream>

#include <sstream>

TaskManager::TaskManager() {}                                                               //Konstruktor des TaskManagers

void TaskManager::addTask(const Task& task) {                                               //Fügt neue Aufgabe zur Aufgabenliste hinzu

    v\_tasks.push\_back(task);

}

void TaskManager::deleteTask(int index) {                                                   //Löscht eine Aufgabe an der angegebenen Indexposition

    if (index >= 0 && index < v\_tasks.size()) {

        v\_tasks.erase(v\_tasks.begin() + index);

    } else {

        cout << "Index error beim Versuch, die Aufgabe zu löschen!" << endl;

    }

}

void TaskManager::markiereAufgabeAlsErledigt(int index) {                                   //Markiert Aufgabe als erledigt (istErledigt Attribut = true)

    if (index >= 0 && index < v\_tasks.size()) {

        v\_tasks[index].setIstErledigt(true);

    } else {

        cout << "Index error beim Versuch, die Aufgabe als erledigt zu markieren!" << endl;

    }

}

vector<Task>& TaskManager::getTasks() {                                                     //gibt (nicht const) Vektor aller Aufgaben zurück

    return v\_tasks;

}

const vector<Task>& TaskManager::getTasks() const {                                         //gibt einen const Vektor aller Aufgaben zurück

    return v\_tasks;

}

void TaskManager::saveAufgaben() {                                                          //speichern aller AUfgaben in die CSV Datei (Persistenzschicht)

    ofstream TasksFile("TasksFile.csv");

    if (TasksFile.is\_open()) {

        TasksFile << "Index;Titel;Beschreibung;Faelligkeitsdatum;Erledigt;" << endl;

        for (const auto& t : v\_tasks) {

            TasksFile << t.getIndex() << ";" << t.getTitel() << ";" << t.getBeschreibung()

                      << ";" << t.getFaelligkeitsdatum() << ";" << t.getIstErledigt() << ";" << endl;

        }

    } else {

        cout << "Datei konnte nicht für das Speichern geöffnet werden!" << endl;

    }

    TasksFile.close();

}

void TaskManager::loadAufgaben() {                                                          //Aufgaben aus der csv-Datei lesen und AUfgabenliste befüllen

    v\_tasks.clear();

    string line;

    ifstream TasksFile("TasksFile.csv");

    if (TasksFile.is\_open()) {

        getline(TasksFile, line); // Header überspringen

        while (getline(TasksFile, line)) {

            stringstream ss(line);

            string tempIndex, titel, beschreibung, faelligkeitsdatum, tempIstErledigt;

            getline(ss, tempIndex, ';');

            getline(ss, titel, ';');

            getline(ss, beschreibung, ';');

            getline(ss, faelligkeitsdatum, ';');

            getline(ss, tempIstErledigt, ';');

            int index = stoi(tempIndex);

            bool istErledigt = (tempIstErledigt == "1" || tempIstErledigt == "true");

            v\_tasks.emplace\_back(TaskFactory::createTask(index, titel, beschreibung, faelligkeitsdatum, istErledigt));

        }

    } else {

        cout << "Datei konnte beim Laden nicht geöffnet werden!" << endl;

    }

    TasksFile.close();

}

int TaskManager::getNextIndex() {

    return v\_tasks.size();                                                                              //Tatsächlicher Index der auch nach Löschen von Aufgaben stimmt

}

                                                                                                        //Attribute einer vorhandenen AUfgabe bearbeiten

void TaskManager::editAufgabe(int index, const std::string& neuerTitel, const std::string& neueBeschreibung, const std::string& neuesFaelligkeitsdatum)

{

    if (index >= 0 && index < v\_tasks.size())

    {

        v\_tasks[index].setTitel(neuerTitel);

        v\_tasks[index].setBeschreibung(neueBeschreibung);

        v\_tasks[index].setFaelligkeitsdatum(neuesFaelligkeitsdatum);

    }

}

Datei „ViewInterface.h“

#pragma once

#include <vector>

#include <string>

#include "Task.h"

class ViewInterface {

public:

    virtual void zeigeAufgaben(const std::vector<Task>& aufgaben) = 0;  //Methode zum Anzeigen aller Aufgaben

    virtual void zeigeFehlermeldung(const std::string& meldung) = 0;    //Methode zum Anzeigen einer Fehlermeldung

    virtual ~ViewInterface() = default;

};

Datei „test\_simple.cpp“

#include <gtest/gtest.h>

#include "TaskManager.h"

#include "Task.h"

#include "TaskFactory.h"

//T01 – TaskFactory erstellt korrekt initialisiertes Task-Objekt

TEST(UnitTest\_TaskFactory, T01\_CreateTaskReturnsCorrectAttributes) {

    //Testdaten definieren

    int index = 1;

    std::string titel = "Testtitel";

    std::string beschreibung = "Testbeschreibung";

    std::string faelligkeitsdatum = "2025-12-31";

    bool istErledigt = false;

    //Task über die Factory erzeugen

    Task task = TaskFactory::createTask(index, titel, beschreibung, faelligkeitsdatum, istErledigt);

    //Alle Attribute müssen den eingegebenen Werten entsprechen

    EXPECT\_EQ(task.getIndex(), index);

    EXPECT\_EQ(task.getTitel(), titel);

    EXPECT\_EQ(task.getBeschreibung(), beschreibung);

    EXPECT\_EQ(task.getFaelligkeitsdatum(), faelligkeitsdatum);

    EXPECT\_EQ(task.getIstErledigt(), istErledigt);

}

//T02 – Hinzufügen vergrößert die Aufgabenliste um eins

TEST(UnitTest\_TaskManagerFunktionen, T02\_AddTaskIncreasesCount) {

    TaskManager manager;

    std::size\_t size\_before = manager.getTasks().size();

    //Eine neue Aufgabe erzeugen (Index entspricht size\_before)

    Task neueAufgabe = TaskFactory::createTask(

        static\_cast<int>(size\_before), //static cast, da size\_before ein size\_t ist

        "Testaufgabe",

        "Beschreibung",

        "2025-12-31",

        false

    );

    //Aufgabe hinzufügen

    manager.addTask(neueAufgabe);

    //Aufgabenanzahl ist um 1 größer

    std::size\_t size\_after = manager.getTasks().size();

    EXPECT\_EQ(size\_after, size\_before + 1);

}

//T03 – Entfernen einer Aufgabe verkleinert die Liste

TEST(UnitTest\_TaskManagerFunktionen, T03\_DeleteTaskReducesCount) {

    TaskManager manager;

    //Zwei Aufgaben hinzufügen

    manager.addTask(TaskFactory::createTask(0, "Aufgabe1", "Desc1", "2025-12-31", false));

    manager.addTask(TaskFactory::createTask(1, "Aufgabe2", "Desc2", "2025-12-31", false));

    std::size\_t size\_before = manager.getTasks().size();

    //Erste Aufgabe löschen

    manager.deleteTask(0);

    std::size\_t size\_after = manager.getTasks().size();

    //Liste ist um 1 Element kleiner

    EXPECT\_EQ(size\_after, size\_before - 1);

}

//T04 – Bearbeiten ändert die Eigenschaften einer Aufgabe

TEST(UnitTest\_TaskManagerFunktionen, T04\_EditTaskChangesAttributes) {

    TaskManager manager;

    //Eine Aufgabe hinzufügen

    manager.addTask(TaskFactory::createTask(0, "AltTitel", "AltDesc", "2025-10-10", false));

    std::size\_t size\_before = manager.getTasks().size();

    //Neue Werte zum Bearbeiten

    std::string neuerTitel = "NeuTitel";

    std::string neueBeschreibung = "NeuDesc";

    std::string neuesDatum = "2025-12-31";

    //Aufgabe an Index 0 bearbeiten

    manager.editAufgabe(0, neuerTitel, neueBeschreibung, neuesDatum);

    //Aufgabe hat neue Werte

    const auto& tasks = manager.getTasks();

    ASSERT\_EQ(tasks.size(), size\_before); // Größe bleibt gleich

    EXPECT\_EQ(tasks[0].getTitel(), neuerTitel);

    EXPECT\_EQ(tasks[0].getBeschreibung(), neueBeschreibung);

    EXPECT\_EQ(tasks[0].getFaelligkeitsdatum(), neuesDatum);

}

//T05 – Markieren einer Aufgabe als erledigt setzt Status auf true

TEST(UnitTest\_TaskManagerFunktionen, T05\_MarkTaskAsDone) {

    TaskManager manager;

    //Eine Aufgabe hinzufügen, die noch nicht erledigt ist

    manager.addTask(TaskFactory::createTask(0, "Beispiel", "Beschreibung", "2025-12-31", false));

    //Aufgabe an Index 0 als erledigt markieren

    manager.markiereAufgabeAlsErledigt(0);

    //Status 'istErledigt' ist jetzt true

    const auto& tasks = manager.getTasks();

    ASSERT\_EQ(tasks.size(), 1u);

    EXPECT\_TRUE(tasks[0].getIstErledigt());

}

// T06 – Speichern und Laden von Aufgaben über CSV-Datei

TEST(UnitTest\_TaskManagerPersistenz, T06\_SaveAndLoad) {

    //Ursprünglicher Manager mit zwei Aufgaben

    TaskManager manager;

    manager.addTask(TaskFactory::createTask(0, "Titel1", "Beschr1", "2025-12-31", false));

    manager.addTask(TaskFactory::createTask(1, "Titel2", "Beschr2", "2025-12-31", true))

    //Aufgaben in CSV-Datei speichern

    manager.saveAufgaben(); //schreibt alle Aufgaben in "TasksFile.csv"

    //Neuer Manager simuliert Programmneustart – lädt Aufgaben aus Datei

    TaskManager neuerManager;

    neuerManager.loadAufgaben(); //liest Aufgaben aus "TasksFile.csv" und füllt Liste

    //Anzahl der geladenen Aufgaben muss übereinstimmen

    const auto& loadedTasks = neuerManager.getTasks();

    ASSERT\_EQ(loadedTasks.size(), 2u);

    //Überprüfung der Attribute der ersten Aufgabe

    EXPECT\_EQ(loadedTasks[0].getIndex(), 0);

    EXPECT\_EQ(loadedTasks[0].getTitel(), "Titel1");

    EXPECT\_EQ(loadedTasks[0].getBeschreibung(), "Beschr1");

    EXPECT\_EQ(loadedTasks[0].getFaelligkeitsdatum(), "2025-12-31");

    EXPECT\_FALSE(loadedTasks[0].getIstErledigt());

    //Überprüfung der Attribute der zweiten Aufgabe

    EXPECT\_EQ(loadedTasks[1].getIndex(), 1);

    EXPECT\_EQ(loadedTasks[1].getTitel(), "Titel2");

    EXPECT\_EQ(loadedTasks[1].getBeschreibung(), "Beschr2");

    EXPECT\_EQ(loadedTasks[1].getFaelligkeitsdatum(), "2025-12-31");

    EXPECT\_TRUE(loadedTasks[1].getIstErledigt());

}

//========================================================================

//MAIN

//========================================================================

int main(int argc, char \*\*argv)

{

    ::testing::InitGoogleTest(&argc, argv);

    return RUN\_ALL\_TESTS();

}

Datei „dialogtaskbearbeiten.h“

#ifndef DIALOGTASKBEARBEITEN\_H

#define DIALOGTASKBEARBEITEN\_H                                  //Header Guard

#include <QDialog>                                              //Implementierung von Dialogfenster

namespace Ui {

class DialogTaskBearbeiten;

}

class DialogTaskBearbeiten : public QDialog                     //Dialogfenster zum bearbeiten der ausgewählten Aufgabe, erbt aus QDialog

{

    Q\_OBJECT

public:

    explicit DialogTaskBearbeiten(QWidget \*parent = nullptr);   //Konstruktor Dialogfenster

    ~DialogTaskBearbeiten();                                    //Destruktor, gibt Speicher frei

    void setTitel(const QString& titel);                        //Methode: um später bereits vorhandene Werte in die Felder des Dialogs zu laden

    void setBeschreibung(const QString& beschreibung);          //Methode: um später bereits vorhandene Werte in die Felder des Dialogs zu laden

    void setFaelligkeitsdatum(const QDate& faelligkeitsdatum);  //Methode: um später bereits vorhandene Werte in die Felder des Dialogs zu laden

private:

    Ui::DialogTaskBearbeiten \*ui;

};

#endif // DIALOGTASKBEARBEITEN\_H

Datei „dialogtaskbearbeiten.cpp“

#include "dialogtaskbearbeiten.h"

#include "ui\_dialogtaskbearbeiten.h"

DialogTaskBearbeiten::DialogTaskBearbeiten(QWidget \*parent)                             //Konstruktor um Task zu bearbeiten

    : QDialog(parent)

    , ui(new Ui::DialogTaskBearbeiten)

{

    ui->setupUi(this);

    connect(ui->AufgabeSpeichernDialog, &QPushButton::clicked, this, &QDialog::accept); //Slot für Button "Aufgabe Speichern" auf Dialog

    connect(ui->AbbrechenDialog, &QPushButton::clicked, this, &QDialog::reject);        //Slot für Button "Abbrechen" auf Dialog

}

DialogTaskBearbeiten::~DialogTaskBearbeiten()                                           //Destruktor

{

    delete ui;

}

void DialogTaskBearbeiten::setTitel(const QString& titel)                               //Setter Methode für Titel

{

    ui->titel->setText(titel);                                                          //schreibt Text des bisherigen Titels in das Eingabefeld

}

void DialogTaskBearbeiten::setBeschreibung(const QString& beschreibung)                 //Setter Methode für Beschreibung

{

    ui->beschreibung->setPlainText(beschreibung);                                       //schreibt Text der bisherigen Beschreibung in das Eingabefeld

}

void DialogTaskBearbeiten::setFaelligkeitsdatum(const QDate& faelligkeitsdatum)         //Setter Methode für Datum

{

    ui->faelligkeitsdatum->setDate(faelligkeitsdatum);                                  //Schreibt bsiher eingestelltes Datum in das Datumsfeld

}

Datei „dialogtaskhinzufuegen.h“

#ifndef DIALOGTASKHINZUFUEGEN\_H

#define DIALOGTASKHINZUFUEGEN\_H

#include <QDialog>

namespace Ui {

class DialogTaskHinzufuegen;

}

class DialogTaskHinzufuegen : public QDialog                        //Dialogfenster zum hinzufügen einer neuen Aufgabe

{

    Q\_OBJECT

public:

    explicit DialogTaskHinzufuegen(QWidget \*parent = nullptr);      //Konstruktor

    ~DialogTaskHinzufuegen();                                       //Destruktor

private slots:

private:

    Ui::DialogTaskHinzufuegen \*ui;

};

#endif // DIALOGTASKHINZUFUEGEN\_H

Datei „dialogtaskhinzufuegen.cpp“

#include "dialogtaskhinzufuegen.h"

#include "ui\_dialogtaskhinzufuegen.h"

DialogTaskHinzufuegen::DialogTaskHinzufuegen(QWidget \*parent)                           //Konstruktor

    : QDialog(parent)

    , ui(new Ui::DialogTaskHinzufuegen)

{

    ui->setupUi(this);

    connect(ui->AufgabeSpeichernDialog, &QPushButton::clicked, this, &QDialog::accept); //Slot für Button "Aufgabe Speichern" auf Dialog

    connect(ui->AbbrechenDialog, &QPushButton::clicked, this, &QDialog::reject);        //Slot für Button "Abbrechen" auf Dialog

}

DialogTaskHinzufuegen::~DialogTaskHinzufuegen()                                         //Destruktor

{

    delete ui;

}

Datei „erledigteaufgabenliste.h“

#ifndef ERLEDIGTEAUFGABENLISTE\_H

#define ERLEDIGTEAUFGABENLISTE\_H

#include <QDialog>

namespace Ui {

class erledigteAufgabenliste;

}

class erledigteAufgabenliste : public QDialog                       //Dialogfenster der erledigten Aufgaben

{

    Q\_OBJECT

public:

    void addTaskToList(const QString& text);                        //Methode um Aufgaben in die Liste einzufügen

    explicit erledigteAufgabenliste(QWidget \*parent = nullptr);     //Kosntruktor

    Ui::erledigteAufgabenliste \*getUi() {return ui; }

    ~erledigteAufgabenliste();                                      //Destruktor

private:

    Ui::erledigteAufgabenliste \*ui;

};

#endif // ERLEDIGTEAUFGABENLISTE\_H

Datei „erledigteaufgabenliste.cpp“

#include "erledigteaufgabenliste.h"

#include "ui\_erledigteaufgabenliste.h"

erledigteAufgabenliste::erledigteAufgabenliste(QWidget \*parent)     //Konstruktor

    : QDialog(parent)

    , ui(new Ui::erledigteAufgabenliste)

{

    ui->setupUi(this);

}

erledigteAufgabenliste::~erledigteAufgabenliste()                   //Destruktor

{

    delete ui;

}

void erledigteAufgabenliste::addTaskToList(const QString& text){    //Methode: um erledigte Aufgabe aus Aufgabenliste in die Liste zu verschieben (Referenz um nicht jedes mal Kopieren zu müssen)

    ui->listWidgetErledigteAufgaben->addItem(text);                 //Zeigt text als neuen Eintrag in der Liste der erledigten Aufgaben an

}

Datei „mainwindow.h“

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QListWidget>

#include <QMainWindow>

#include <QVector>

#include "../TaskManager.h"           //Einbinden der eigenen Klasse zum verwalten der Aufgaben

#include "../TaskController.h"

#include "../ViewInterface.h"

QT\_BEGIN\_NAMESPACE

namespace Ui {

class MainWindow;

}

QT\_END\_NAMESPACE

class MainWindow : public QMainWindow, public ViewInterface

{

    Q\_OBJECT

public:

    void zeigeAufgaben(const std::vector<Task>& aufgaben) override;

    void zeigeFehlermeldung(const std::string& meldung) override;

    MainWindow(QWidget \*parent = nullptr);                                      //Konstruktor des MainWindows

    ~MainWindow();                                                              //Destruktor

private slots:

    void on\_pBAufgabeHinzufuegen\_clicked();                                     //Slot auf den Button, mit dem eine Aufgabe hinzugefügt wir

    void on\_pBAufgabeErledigt\_clicked();                                        //Slot auf den Button, mit dem eine Aufgabe als erledigt markiert wird

    void on\_pBErledigteA\_clicked();                                             //Slot auf den Button, mit dem der Dialog/die Liste für die erledigten Aufgaben geöffnet wird

    void on\_pBUeberfaelligeA\_clicked();                                         //Slot auf den Button, mit dem der Dialog/die Liste der überfälligen Aufgaben geöfnet wird

    void on\_pBAufgabeEntfernen\_clicked();                                       //Slot auf den Button, mit dem eine ausgewählte Aufgabe entfernt werden kann

    void on\_listWidget\_itemDoubleClicked(QListWidgetItem \*item);                //Slot der die Bearbeitung durch einen Doppelklick ermöglicht

private:

    Ui::MainWindow \*ui;

    TaskManager taskManager;

    TaskController\* taskController;

    QVector<int> guiToVectorIndex;                                              //Zuordnungsliste zwischen den QListWidget einträgen und tatsächlichen Aufgaben Indizes

};

#endif // MAINWINDOW\_H

Datei „mainwindow.cpp“

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include "dialogtaskbearbeiten.h"

#include "dialogtaskhinzufuegen.h"

#include "erledigteaufgabenliste.h"

#include "ueberfaelligeaufgaben.h"

#include <QLineEdit>

#include <QDateEdit>

#include <QTextEdit>

#include <QMessageBox>

MainWindow::MainWindow(QWidget \*parent)                                                         //Konstruktor

    : QMainWindow(parent)

    , ui(new Ui::MainWindow)

{

    ui->setupUi(this);

    taskController = new TaskController(&taskManager);                                          //dem controller das Modell (taskManager) übergeben

    taskController->load();                                                                     //Aufgaben beim Starten der Anwendung laden

    zeigeAufgaben(taskController->getTasks());;                                                                        //Aktualisieren von Mapping & ListWidget

    ui->listWidget->clear();                                                                    //sicherstellen, dass Anzeige-Liste leer ist (nur GUI, nicht gespeicherten Aufgaben)

    guiToVectorIndex.clear();                                                                   //Mapping leeren

    const auto& tasks = taskController->getTasks();                                             //alle gespeicherten Aufgaben holen

    for (int i = 0; i < tasks.size(); ++i)                                                      //Schleife über alle Aufgaben

    {

        const Task& t = tasks[i];

        if(!t.getIstErledigt())                                                                 //nur die offenen Aufgaben im mainwindow zeigen

        {

            QDate faellig = QDate::fromString(QString::fromStdString(t.getFaelligkeitsdatum()), "yyyy-MM-dd");

            QString statusText = (faellig.isValid() && faellig < QDate::currentDate()) ? "Überfällig" : "Offen";

            QString itemText = QString::fromStdString(t.getTitel()) + " | "

                               + QString::fromStdString(t.getBeschreibung()) + " | "

                               + QString::fromStdString(t.getFaelligkeitsdatum()) + " | "

                               + statusText;

            ui->listWidget->addItem(itemText);                                                  //Attribute jedes Tasks schreiben

            guiToVectorIndex.append(i);                                                         //Mapping der noch nicht erledigten Aufgaben für List MainWindow (sonst würden auch erledigte

        }                                                                                           //Indizes angezeigt werden)

    }

}

MainWindow::~MainWindow()                                                                       //Destruktor

{

    taskController->save();                                                                    //speichern vor dem beenden der Anwendung

    delete ui;

}

void MainWindow::on\_pBAufgabeHinzufuegen\_clicked()                                              //Methode: Logik für das hinzufügen einer Aufgabe (=extra Dialog)

{

    DialogTaskHinzufuegen dialog(this);                                                         //Erstellt Dialogfenster, setzt MainWindow als Elternfenster (=this) damit darüber angezeigt

    if (dialog.exec() == QDialog::Accepted)                                                     //wenn Dialog ausgeführt ist

    {

        QString titel = dialog.findChild<QLineEdit\*>("titel")->text();                          //liest Inhalt aus Eingabefeld im Dialog

        QDate faelligkeitsdatum = dialog.findChild<QDateEdit\*>("faelligkeitsdatum")->date();    //liest Inhalt aus Eingabefeld im Dialog

        QString beschreibung = dialog.findChild<QTextEdit\*>("beschreibung")->toPlainText();     //liest Inhalt aus Eingabefeld im Dialog

        if(titel.isEmpty())                                                                     //Eingabevalidierung für den Titel gemäß Anforderungsdokument

        {

            QMessageBox::warning(this, "Fehler", "Titel ist leer!");

            return;

        };

        if(faelligkeitsdatum < QDate::currentDate())                                            //Eingabevalidierung für Datum gemäß Anforderungsdokument

        {

            QMessageBox::warning(this, "Fehler", "Datum liegt in der Vergangenheit!");

            return;

        };

        taskController->addAufgabe(titel.toStdString(), beschreibung.toStdString(), faelligkeitsdatum.toString("yyyy-MM-dd").toStdString());

        zeigeAufgaben(taskController->getTasks());;

    }

}

void MainWindow::on\_pBAufgabeErledigt\_clicked()                                                 //Methode: Logik für Aufgabe erledigt

{

    int guiIndex = ui->listWidget->currentRow();                                                //holt aktuell in Liste ausgewählte Aufgabe

    if(guiIndex >= 0)

    {

        int realIndex = guiToVectorIndex.at(guiIndex);                                          //Mapping auf echten Vector-Index!

        taskController->markiereAufgabeAlsErledigt(realIndex);

        delete ui->listWidget->takeItem(guiIndex);                                              //Aus der Liste (MainWindow) entfernen

        zeigeAufgaben(taskController->getTasks());;

    }

}

void MainWindow::on\_pBErledigteA\_clicked()                                                      //Methode: Logik für Liste der erledigten Aufgabenm

{

    erledigteAufgabenliste dialog(this);                                                        //Dialogfenster erstellen

    const auto& tasks = taskController->getTasks();

    for(const Task& t : tasks) {                                                                //Iteriert über alle Element

        if (t.getIstErledigt()) {

            dialog.addTaskToList(QString::fromStdString(t.getTitel()));                         //wenn erledigt, wird Aufgabe in die erledigt Liste eingefügt

        }

    }

    dialog.exec();                                                                              //Dialogfenster öffnen

}

void MainWindow::on\_pBUeberfaelligeA\_clicked()                                                  //Methode: Logik für Liste der überfälligen Aufgaben

{

    ueberfaelligeAufgaben dialog(this);                                                         //Dialog erstellen

    const auto& tasks = taskController->getTasks();                                                //vector mit Tasks

    QDate today = QDate::currentDate();                                                         //aktuelles Datum

    for(const Task& t : tasks)                                                                  //iteriert über Aufgaben

    {

        QDate faellig = QDate::fromString(QString::fromStdString(t.getFaelligkeitsdatum()), "yyyy-MM-dd");//holt Attribut Faelligkeitsdatum

        if(!t.getIstErledigt() && faellig.isValid() && faellig < today)                         //prüft ob noch nicht erledigt, faellig und mindestens von gestern

        {

            dialog.addTaskToList(QString::fromStdString(t.getTitel()));                         //mit Methode in Liste ueberfaelliger Aufgaben einfügen

        }

    }

    dialog.exec();

}

void MainWindow::on\_pBAufgabeEntfernen\_clicked()                                                //Methode: Aufgabe entfernen (komplett löschen)

{

    QListWidgetItem\* item = ui->listWidget->currentItem();                                      //Referenz auf ausgewählte Aufgabe in Liste

    if(item)

    {

        int guiIndex = ui->listWidget->currentRow();

        int realIndex = guiToVectorIndex.at(guiIndex);

        if(guiIndex >= 0)

        {

            taskController->delAufgabe(realIndex);

            zeigeAufgaben(taskController->getTasks());;

        }

    }

}

void MainWindow::on\_listWidget\_itemDoubleClicked(QListWidgetItem \*item)                         //Methode: bearbeiten einer Aufgabe in mainWindow

{

    int guiIndex = ui->listWidget->row(item);                                                   //holt ausgewählte Aufgabe aus Liste

    if (guiIndex < 0 || guiIndex >= guiToVectorIndex.size())

    {

        return;

    };

    int realIndex = guiToVectorIndex.at(guiIndex);                                              //Mapping

    const Task& t = taskController->getTasks().at(realIndex);                                   //ausgewählte Aufgabe mit tatsächlichen Index holen

    DialogTaskBearbeiten dialog(this);                                                          //Bearbeiten Dialog erstellen (this = mainwindow)

    dialog.setTitel(QString::fromStdString(t.getTitel()));                                      //bisherige Attribute holen

    dialog.setBeschreibung(QString::fromStdString(t.getBeschreibung()));

    dialog.setFaelligkeitsdatum(QDate::fromString(QString::fromStdString(t.getFaelligkeitsdatum()), "yyyy-MM-dd"));

    if (dialog.exec() == QDialog::Accepted)

    {

        QString neuerTitel = dialog.findChild<QLineEdit\*>("titel")->text();                     //neuen Attribute im Eingabefeld schreiben

        QString neueBeschreibung = dialog.findChild<QTextEdit\*>("beschreibung")->toPlainText(); //Zeiger auf das Widget (benötigt findChild)

        QDate neuesDatum = dialog.findChild<QDateEdit\*>("faelligkeitsdatum")->date();

        taskController->editAufgabe(realIndex, neuerTitel.toStdString(), neueBeschreibung.toStdString(), neuesDatum.toString("yyyy-MM-dd").toStdString());

        zeigeAufgaben(taskController->getTasks());;

    }

}

void MainWindow::zeigeAufgaben(const std::vector<Task>& aufgaben)

{

    ui->listWidget->clear();

    guiToVectorIndex.clear();

    QDate today = QDate::currentDate();

    for (int i = 0; i < aufgaben.size(); ++i)

    {

        const Task& t = aufgaben[i];

        if(!t.getIstErledigt())

        {

            QDate faellig = QDate::fromString(QString::fromStdString(t.getFaelligkeitsdatum()), "yyyy-MM-dd");

            QString statusText = (faellig.isValid() && faellig < today) ? "Überfällig" : "Offen";

            QString itemText = QString::fromStdString(t.getTitel()) + " | "

                               + QString::fromStdString(t.getBeschreibung()) + " | "

                               + QString::fromStdString(t.getFaelligkeitsdatum()) + " | "

                               + statusText;

            ui->listWidget->addItem(itemText);

            guiToVectorIndex.append(i);

        }

    }

}

void MainWindow::zeigeFehlermeldung(const std::string& meldung)

{

    QMessageBox::warning(this, "Fehler", QString::fromStdString(meldung));

}

Datei “ueberfaelligeaufgaben.h”

#ifndef UEBERFAELLIGEAUFGABEN\_H

#define UEBERFAELLIGEAUFGABEN\_H

#include <QDialog>

namespace Ui {

class ueberfaelligeAufgaben;

}

class ueberfaelligeAufgaben : public QDialog                        //Klasse für den Dialog der überfälligen Aufgaben

{

    Q\_OBJECT

public:

    void addTaskToList(const QString& text);                        //Methode: Fügt Aufgabe in die Liste der erledigten Aufgaben (nachdem Datum überschritten)

    explicit ueberfaelligeAufgaben(QWidget \*parent = nullptr);      //Konstruktor

    ~ueberfaelligeAufgaben();                                       //Destruktor

private:

    Ui::ueberfaelligeAufgaben \*ui;

};

#endif // UEBERFAELLIGEAUFGABEN\_H

Datei „ueberfaelligeaufgaben.cpp“

#include "ueberfaelligeaufgaben.h"

#include "ui\_ueberfaelligeaufgaben.h"

ueberfaelligeAufgaben::ueberfaelligeAufgaben(QWidget \*parent)

    : QDialog(parent)

    , ui(new Ui::ueberfaelligeAufgaben)

{

    ui->setupUi(this);

}

ueberfaelligeAufgaben::~ueberfaelligeAufgaben()                         //Destruktor

{

    delete ui;

}

void ueberfaelligeAufgaben::addTaskToList(const QString& text)          //Methode für hinzufügen in ueberfällige Liste

{

    ui->listWidgetUeberfaellige->addItem(text);                         //text als neuer Eintrag in der Liste

}