HAW HAMBURG

BACHELOR THESIS
Akif Aydin

Modernisierung einer bestehenden Anwendung: Ein KI-unterstützter Ansatz zur GUI-Migration

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Akif Aydin

Modernisierung einer bestehenden Anwendung: Ein KI-unterstützter Ansatz zur GUI-Migration

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

Betreuender Prüfer: Prof. Dr.-Ing. Lars Hamann
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 21. Juli 2025

Akif Aydin

Thema der Arbeit

Modernisierung einer bestehenden Anwendung: Ein KI-unterstützter Ansatz zur GUI-Migration

Stichworte

Java, Migration, Swing, JavaFX, AWT, GUI, SwingNode, KI

Kurzzusammenfassung

Diese Thesis thematisiert einen KI-unterstützten Ansatz zur Modernisierung einer bestehenden Anwendung. Hierbei werden Techniken sowie Migrationsstrategien zur Softwaremodernisierung untersucht. Der ganze Prozess der Softwareentwicklung wird mit der KI ChatGPT begleitet.

Akif Aydin

Title of Thesis

Modernizing an Existing Application: An AI-Assisted Approach to GUI-Migration

Keywords

Java, Migration, Swing, JavaFX, AWT, GUI, SwingNode, AI

Abstract

This thesis addresses an AI-supported approach to modernizing an existing application. It examines techniques and migration strategies for software modernization. The entire software development process is supported by the AI ChatGPT.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

AWT	Abstract Window Toolkit.
CSS	Cascading Style Sheets.
GPU	Graphics Processing Unit.
GUI	Graphical User Interface.
JDK	Java Development Kit.
KI	Künstliche Intelligenz.
LOC	Lines Of Code.
MVC	Model-View-Controller.
OCL	Object Constraint Language.
SDK	Software Development Kit.
SoC	Separation of Concerns.
UI	User Interface.
UML	Unified Modeling Language.
USE	UML-basierte Spezifikationsumgebung (UML-based Specification Environment).
XML	Extensible Markup Language.

Listings

1 Einleitung

„There is an urgent need to develop automated techniques to modernize old code.“[17, S. 2514]. Mit diesem Satz verdeutlicht Nitin die Aktualität und Relevanz der Modernisierung einer bestehenden Anwendung: Ein KI-unterstützter Ansatz zur GUI-Migration.

In diesem Kapitel werden die Motivation und die Zielsetzung der Arbeit dargestellt. Darüber hinaus wird die Struktur der Arbeit vorgestellt, um dem Leser einen ersten Überblick darüber zu geben, was ihn erwartet und wie der Weg zum Ziel aufgebaut ist.

1.1 Motivation

„Old code is prone to security vulnerabilities, and is difficult to maintain and upgrade.“ [17, S. 2514]. Aus diesem Satz geht hervor, wieso die Modernisierung einer bestehenden Anwendung von Relevanz ist.

Die Motivation für die Migration einer bestehenden Anwendung von Swing/AWT zu JavaFX sowie die folgenden Informationen stammen aus [20, S. 1]. Die Verwendung von Swing und AWT ist in Legacy-Anwendungen weit verbreitet. Diese beiden GUI-Technologien stammen aus den späten 1990er Jahren. Genauer gesagt wurde das Abstract Window Toolkit (AWT) im Jahr 1995 veröffentlicht, und Swing ist seit 1998 Teil der Java Standard Edition 1.2. Im Vergleich zu diesen beiden älteren GUI-Technologien stammt JavaFX aus dem Jahr 2008, außerdem ist es seit 2011 Open Source. Seit 2018 ist es nicht mehr Teil des Java-JDK, sondern ein Teil des OpenJDK im Projekt OpenJFX, um die Entwicklungsgeschwindigkeit zu erhöhen. Der Vorteil von JavaFX im Vergleich zu den vorherigen ist die Möglichkeit zur Nutzung von Cascading Style Sheets (CSS)-Styling, FXML und der Scene Builder. Der Scene Builder ermöglicht die leichtere Erstellung von GUIs über eine visuelle Entwicklungsumgebung mithilfe von Drag and Drop. Die im Scene Builder erstellte GUI wird als FXML-Datei gespeichert. Neben diesen Vorteilen

besitzt JavaFX eine Graphics-Rendering-Pipeline, von der behauptet wird, dass sie eine bessere Leistung für Graphics Processing Units (GPUs) anbietet. Diese Vorteile führen dazu, dass Anwendungen, welche Swing und AWT verwenden, in JavaFX umgeschrieben werden.

„Künstliche Intelligenz (KI) hat sich zu einem der bedeutendsten und revolutionärsten Bereiche der Technologie entwickelt, insbesondere in der Softwareentwicklung. Sie verspricht, die Art und Weise, wie Entwickler Software entwerfen, entwickeln und einsetzen, grundlegend zu verändern.“, schreibt Philipp Erler in seinem Blogpost [4] und zeigt somit die wichtige Rolle der KI in der heutigen Softwareentwicklung und wie sie diese verändert.

Die Bedeutung der KI in der heutigen Zeit und das Streben nach der Modernisierung einer Legacy-Anwendung aufgrund diverser Vorteile war die Motivation zur Zusammenführung beider Punkte. Somit wurde in dieser Arbeit zur Unterstützung der Migration in eine modernere Technologie der KI-Chatbot ChatGPT verwendet.

1.2 Zielsetzung der Arbeit

Ziel dieser Arbeit ist es, eine bestehende Legacy-Anwendung mithilfe von ChatGPT auf eine moderne GUI-Technologie zu migrieren. Hierbei liegt der Fokus auf der allgemeinen Herangehensweise für solch einen Migrationsansatz, damit diverse Risiken und Kosten bei solch einer Modernisierung minimiert werden können.

Es werden geeignete Migrationsstrategien, Werkzeuge und Techniken analysiert, damit alle Funktionalitäten der Anwendung auch nach der Migration erhalten bleiben und nicht verloren gehen.

ChatGPT wird als unterstützendes Tool verwendet, in der Hoffnung, den Prozess der Modernisierung zu beschleunigen und ein solches umfangreiches Projekt deutlich einfacher und zugänglicher zu machen.

Als Legacy-Anwendung dient in dieser Arbeit die Anwendung USE. Diese Anwendung ist eine UML-basierte Spezifikationsumgebung (UML-based Specification Environment) (USE), mit der Informationssysteme spezifiziert und validiert werden können [25]. In dieser Anwendung wird die GUI-Technologie von Swing/AWT nach JavaFX migriert, um die genannten Ziele zu verwirklichen.

1.3 Zielgruppe

Zielgruppe dieser Arbeit sind Personen, die eine Legacy-Anwendung nutzen und sich dafür entscheiden, diese zu modernisieren. Ihnen soll eine strukturierte Herangehensweise geboten werden, der sie folgen können, um diverse Risiken und Kosten zu minimieren.

1.4 Aufbau der Arbeit

Beginnend mit Kapitel 2 wird die Rolle und der Ablauf einer Softwaremodernisierung vorgestellt. Daraufhin wird gezeigt, welche GUI-Technologien die vorgestellte Legacy-Anwendung verwendet und was die Ziel-GUI für diese Anwendung ist. Ein Vergleich zwischen diesen beiden GUI-Technologien wird durchgeführt. Nach der Vorstellung der GUI-Technologien wird die Rolle der KI in der heutigen Softwareentwicklung vorgestellt, gefolgt von Herausforderungen in der GUI-Migration. Zum Schluss werden verwandte Arbeiten und Forschungsergebnisse vorgestellt.

Im darauffolgenden Kapitel 3 geht es um die Methodiken für die erfolgreiche Durchführung einer Migration der Legacy-Anwendung. Zunächst werden die Vorbedingungen identifiziert, die für eine Migration notwendig sind. Daraufhin wird eine Migrationsstrategie entwickelt, die auf die vorhandene Legacy-Anwendung abgestimmt ist. Zum Schluss werden die verwendeten Werkzeuge und Techniken vorgestellt.

Im Mittelpunkt dieser Arbeit steht Kapitel 4. Hier geht es um das praktische Vorgehen, die Nutzung der zuvor vorgestellten Migrationsstrategie, Werkzeuge und Techniken. Außerdem wird auf die Rolle der KI näher eingegangen.

In den letzten zwei Kapiteln 5 und 6 geht es um die Evaluation und das Fazit dieser Arbeit. Die Evaluation beinhaltet die Reflexion zur Zielsetzung der Arbeit, ob diese erreicht wurde, die Vorteile der neuen GUI gegenüber der alten, die Vorstellung der in der Migration gewonnenen Erkenntnisse und die begegneten Herausforderungen. Das Fazit fasst die Arbeit zusammen und gibt einen Ausblick über die möglichen zukünftigen Arbeiten.

2 Grundlagen

Dieses Kapitel befasst sich mit den Grundlagen, die für diese Arbeit notwendig sind. Beginnend mit der Bedeutung der Softwaremodernisierung, gefolgt von der GUI-Technologie von USE und der Ziel-GUI, gefolgt von einem Vergleich der beiden GUI-Technologien. Anschließend wird die Rolle der KI in der heutigen Softwareentwicklung sowie die Herausforderungen in der GUI-Migration vorgestellt. Zum Schluss werden verwandte Arbeiten und Forschungsergebnisse betrachtet.

2.1 Softwaremodernisierung

„Wenn das Altsystem zur Altlast wird“, schreibt das Frauhofer-Institut für Experimentelles Software Engineering IESE [5]. Die folgenden Informationen stammen aus dem Beitrag des Frauhofer IESE [5].

Ein Altsystem bedeutet ein Legacy-System, welches alt ist und weiterhin in Verwendung ist. Viele Unternehmen besitzen solch ein Altsystem. Diese Systeme sind sehr wichtig für diese Unternehmen, da sie ein fester Bestandteil von ihnen sind und nicht so einfach wegzudenken sind.

Wichtig ist die Aufrechterhaltung solcher Systeme, welche in den meisten Fällen durch die Pflege dieser Systeme erreicht wird. Trotzdem kommt es zu Punkten im Lebenszyklus, in denen die Pflege alleine nicht mehr ausreicht und eine Modernisierung des Systems notwendig wird. Gründe können abgekündigte Kerntechnologien, Sicherheitsprobleme oder vielerlei andere Faktoren sein.

Die Modernisierung eines Legacy-Systems ist eine große Herausforderung und kann komplexer sein als eine Neuentwicklung. Einer von vielen Gründen ist, dass das Altsystem parallel zum neuen System läuft. Daher ist die Migrationsstrategie sehr wichtig, um diesen Übergang in das neue System ermöglichen zu können.

Die oben genannten Informationen in den Absätzen stammen aus dem Beitrag des Fraunhofer IESE [5].

Laut der Studie *A Survey of Legacy System Modernization Approaches* [9, S. 3 f.], lässt sich der Vorgang der Modernisierung in drei Kategorien einteilen. Die erste Kategorie ist die *Wartung* (Maintenance) eines Systems, die eine wichtige Rolle für die Unterstützung der Evolution eines Systems spielt. Sie umfasst nur das Beheben von Bugs und die Erweiterung von Funktionen. Die Wartung von Legacy-Systemen ist jedoch leider sehr kostspielig und steigt mit der Zeit inkrementell. Die zweite Kategorie ist das *Ersetzen* (Replacement) eines Altsystems, welches zum Einsatz kommt, wenn die Dokumentation des Systems unzureichend ist, das System veraltet ist oder das System nicht erweiterbar ist. Das komplette Ersetzen eines Systems ist sehr ressourcenintensiv und außerdem erfordert dieser Vorgang ein sehr umfangreiches Testen. Die letzte Kategorie ist die *Modernisierung* (Modernization) eines Systems, die aufwendiger ist als die Wartung. Ein großer Teil des existierenden Systems wird weiterhin verwendet.

In dieser Arbeit wird die Modernisierung einer bestehenden Anwendung an einer Anwendung durchgeführt, in der die GUI von Swing/AWT in JavaFX migriert wird. In den folgenden Abschnitten wird näher auf die aktuelle GUI der Anwendung, die Ziel-GUI und die Unterschiede der beiden eingegangen.

2.2 Aktuelle GUI-Technologie

In diesem Abschnitt liegt der Fokus auf dem grundlegenden Aufbau der GUI der Legacy-Anwendung.

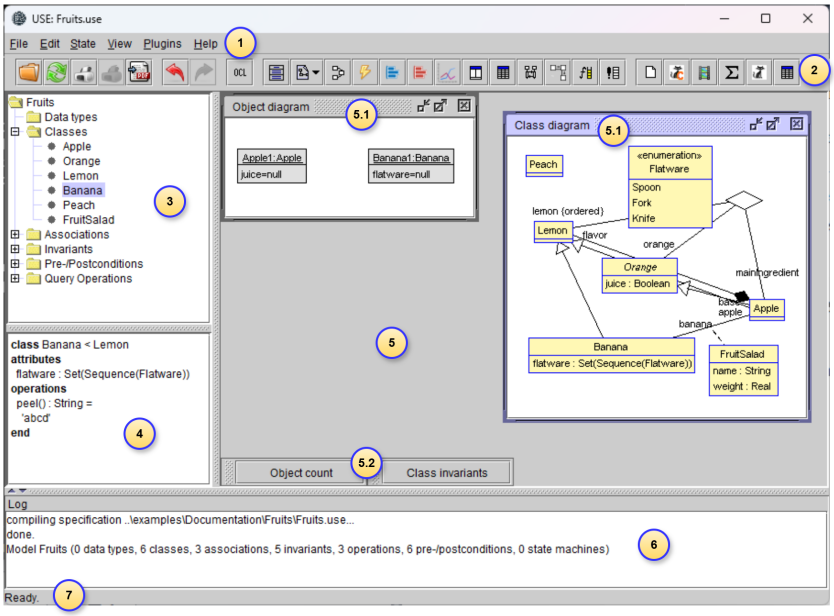


Abbildung 2.1: GUI-Struktur der Legacy-Anwendung USE

In der oberen Abbildung ist die GUI-Struktur der Anwendung USE zu sehen. Diese GUI besitzt mehrere Bereiche, die verschiedene Aufgabenbereiche abdecken. Oben in der Anwendung ist eine Menüleiste (1) zu sehen, und unter der Menüleiste ist eine Toolleiste (2) zu finden, welche die diversen Funktionen visuell darstellt. Auf der linken Seite befindet sich eine Baumstruktur (3), welche die Informationen zum geladenen Modell aufzeigt, und unter dieser ein Informationsfeld (4) zur Anzeige von Informationen für ausgewählte Elemente aus dem Baum. Die rechte Seite der Anwendung beinhaltet ein DesktopPane (5) als ein Multiple-Document-Interface (MDI). Innerhalb dieses DesktopPane können diverse Fenster (5.1) angezeigt werden, in denen verschiedene Informationen wie Klassendiagramme und Objektdiagramme angezeigt werden. Im unteren Bereich des DesktopPanes befindet sich eine Taskleiste (5.2) zum Minimieren und wieder Aufklappen der Fenster. Auf der unteren Seite der Anwendung befindet sich ein Ausgabefeld (6) für die Ausgaben der Anwendung sowie eine Statusleiste (7) für diverse Statusanzeigen.

Die Anwendung verwendet für die GUI Swing und AWT, welche beide sehr alt sind und aus den 1990er Jahren stammen, wie in Abschnitt 1.1 beschrieben wurde. Laut Panitz [19, S. 388] wird Swing nicht alleine, sondern gemeinsam mit AWT verwendet, da die Swing-Klassen von den AWT-Klassen abgeleitet sind. Zudem existieren Klassen, wie zum Beispiel Ereignisklassen, welche die Funktionalität von grafischen Objekten bestimmen,

nicht in Swing, sondern nur in AWT. Deshalb kommen bei der Entwicklung mit Swing auch AWT-Bibliotheken zum Einsatz.

Aus der folgenden Aussage: „Mittlerweile soll langfristig Swing nicht mehr weiterentwickelt werden, sondern in Zukunft die Bibliothek Java FX der Standard zur GUI Programmierung in Java sein.“ [19, S. 387], geht hervor, dass Swing nicht mehr weiterentwickelt wird und sich die Java-GUI-Entwicklung in Richtung JavaFX richtet.

2.3 Ziel-GUI-Technologie

JavaFX ist als OpenJFX ein Teil des OpenJDK, wie in der Motivation in Abschnitt 1.1 beschrieben wurde. Das Besondere hieran ist, dass JavaFX somit ein Open-Source-Projekt ist und eine große Community besitzt, die aktiv am Projekt weiterarbeitet, wie auch im Repository [1] auf GitHub zu sehen ist.

In JavaBeginners.de [14] wird hervorgehoben, dass JavaFX dem Model-View-Controller-Entwurfsmuster (MVC) folgt, da die GUI-Oberfläche mithilfe einer XML-basierten Markup-Sprache FXML von der Programmlogik getrennt werden kann. Zudem wird erwähnt, dass die Verwendung von FXML die Wartbarkeit der GUI-Oberfläche verbessert.

Die FXML-Dateien können beispielsweise mithilfe des Scene Builder erstellt werden, indem man Benutzeroberflächen per Drag-and-Drop ganz einfach erstellen kann [20].

In JavaFX ist das Styling der Benutzeroberfläche deutlich einfacher als zuvor, da die Verwendung von CSS den Entwicklern viele neue Möglichkeiten zur Anpassung der Benutzeroberfläche eröffnet [3].

JavaFX unterstützt außerdem hardwarebeschleunigtes Grafik-Rendering, was sich je nach Anwendungsfall positiv oder negativ auf die Performance auswirken kann [20, S. 1792].

2.4 Vergleich der Technologien

In diesem Abschnitt wird ein Vergleich zwischen Swing/AWT und JavaFX durchgeführt, um die Unterschiede besser darstellen zu können. Die folgenden Informationen für diesen Vergleich stammen aus GeeksforGeeks [2].

Leistungsvergleich: Mit Swing kann eine gute Leistung bei mittlerer bis hoher Komplexität einer Anwendung erreicht werden, denn es ist auf Leistung optimiert und ermöglicht es effizient komplexere Benutzeroberflächen darzustellen. JavaFX bietet eine hervorragende Leistung, wenn es sich um Anwendungen handelt, die erweiterte Grafikfunktionen und Multimediafunktionen beinhalten. Außerdem kann JavaFX Hardwarebeschleunigung verwenden, die zum Rendern von Bildern dient und somit unter anderem leistungsstarke visuelle Effekte erzeugen kann.

Umfangreiche UI-Komponenten: Swing besitzt eine umfangreiche Bibliothek. Die Größe der Bibliothek ermöglicht die Gestaltung komplexer und moderner Benutzeroberflächen. JavaFX besitzt eine moderne Auswahl an UI-Komponenten, dazu gehören unter anderem Diagramme, Mediaplayer und 3D-Objekte. Mithilfe dieser großen Auswahl entstehen viele Möglichkeiten für die Entwicklung funktionsreicher und ansprechender Benutzeroberflächen.

Benutzerfreundlichkeit: Swing ist umfangreich aufgrund der Vielzahl der gebotenen UI-Komponenten, was es für Anfänger komplexer machen kann. JavaFX hingegen bietet im Vergleich zu Swing modernere UI-Komponenten, die ebenfalls komplexer sein können. Dazu kommen weitere verwendbare Technologiemöglichkeiten, wie CSS und FXML. Mithilfe dieser beiden Technologien ermöglicht JavaFX eine modernere und auch flexiblere Entwicklung.

Kompatibilität mit modernem Java: Swing und JavaFX sind beide kompatibel mit modernen Java-Standards. Swing wird weiterhin in modernen Entwicklungsumgebungen unterstützt, wobei jedoch JavaFX in Java-Anwendungen als empfohlene Wahl für die Entwicklung moderner Benutzeroberflächen gilt.

Community-Support und -Wartung: Swing besitzt eine große Menge an Ressourcen und starken Community Support, der weiterhin aktiv ist. JavaFX hat eine wachsende Entwickler-Community und wird aktiv gepflegt. Das empfohlene UI-Toolkit ist JavaFX für moderne Java-Anwendungen, da es regelmäßig verbessert wird und Updates erhält.

Plattformkompatibilität: Swing bietet nur eine eingeschränkte Web- und Mobilgeräteunterstützung, es ist hauptsächlich auf die Entwicklung von Desktop-Anwendungen fokussiert. JavaFX bietet eine viel größere Kompatibilität, da es plattformübergreifend ausgeführt werden kann, unter anderem auf Desktop, Web und Mobilgeräten. JavaFX bietet für die Bereitstellung im Web eine Unterstützung mithilfe von Technologien wie WebView. Für Mobilgeräte ist die Unterstützung noch begrenzt, woran jedoch weiterhin gearbeitet wird, um diese zu verbessern.

Ein großer Faktor für die Migration der GUI von Swing in JavaFX ist laut Robillard und Kutschera [22, S. 78], dass eine Migration nicht umgangen werden kann, um in einer sich immer weiterentwickelnden Welt mit immer neueren und moderneren Hardwareumgebungen relevant zu bleiben.

2.5 KI in der Softwareentwicklung

Wie in Abschnitt 1.1 erwähnt, spielt in der heutigen Softwareentwicklung die Künstliche Intelligenz (KI) eine sehr wichtige Rolle. In diesem Abschnitt wird die Rolle der KI in der Softwareentwicklung anhand des Blogposts *KI in der Softwareentwicklung - Bedeutung und Potenziale* [4] beschrieben.

Künstliche Intelligenz kann für komplexere Aufgaben verwendet werden, wie das Lernen, Planen und das Lösen von diversen Problemen. Mithilfe der KI lassen sich somit besser Zusammenhänge erkennen und Rückschlüsse ziehen.

Die KI bringt viel Potenzial mit sich. Mithilfe der KI lässt sich die Effizienz in der Entwicklung steigern, außerdem können die Entwicklungskosten reduziert werden, da die Möglichkeit existiert, die KI in der Entwicklungsumgebung zu verwenden. Die KI spart Zeit, indem sie bei der Implementierung unterstützen kann und ermöglicht repetitive sowie zeitintensive Aufgaben zu automatisieren.

Mithilfe der KI können Fehler frühzeitig gefunden werden, um Sicherheitsprobleme und Programmierfehler zu vermeiden. In der Qualitätssicherung spielt sie somit eine große Rolle, da sie bei Code-Reviews, beim Testing und in der Fehlererkennung und Fehlerbehebung unterstützen kann. Mithilfe der KI in der Qualitätssicherung bleibt für die Entwickler mehr Zeit für andere, komplexere Arbeiten. Somit können sie ihre Zeit und Ressourcen für wichtigere Aufgaben in der Softwareentwicklung konzentrieren.

In Bereichen wie der Cybersicherheit können KI-Systeme dabei unterstützen, sensible Daten zu schützen, indem sie Bedrohungen erkennen können und somit auch helfen, rechtzeitig auf Sicherheitsvorfälle zu reagieren. Mithilfe der KI können somit langfristig Sicherheitsvorfälle vermieden werden.

Die KI wird den Entwickler nicht ersetzen, sondern es ihn ermöglichen sich auf die kreativeren und komplexeren Aufgaben zu fokussieren. Außerdem wird der Entwickler nicht mehr den kompletten Code schreiben müssen, sondern eher KI-generierten Code prüfen und anpassen. Somit durchleben die Entwickler, kein Aussterben der klassischen Programmierung, sondern eher eine Evolution dessen.

Es gibt verschiedene Tools wie GitHub Copilot, AskCode und viele mehr. Diese Tools werden in verschiedenen Bereichen der modernen Softwareentwicklung eingesetzt. Diese KI-Tools bringen jedoch unterschiedliche Vor- und Nachteile, je nach Einsatzgebiet, mit sich.

Der Einsatz von Künstliche Intelligenz bringt für die Softwareentwicklung einen bedeutsamen Fortschritt mit sich, der die komplette Branche grundlegend verändern könnte. So toll es sich mit der KI auch anhört, darf im letzten Schritt die Überprüfung durch einen Menschen nicht fehlen, da die KI immer noch Fehler machen kann.

2.6 Herausforderungen in der GUI-Migration

Wie in *Lessons Learned While Migrating From Swing to JavaFX* [22, S. 78] beschrieben wurde, stellt sich jedem Projektleiter einer Anwendung mit einem älteren Framework irgendwann die Frage der Migration in ein neues Framework. Diese Frage bringt jedoch Bedenken und Ängste mit sich, da es nicht einfach ist und viele technische Details mit sich bringt, die man kennen muss. Zudem sind die Probleme, die in solch einer großen Migration auftreten können, so gut wie unendlich.

Die Studie [22, S. 80 ff.] führt wichtige Kenntnisse darüber auf, was in einer großen Migration für Herausforderungen auftreten können, und diese Herausforderungen werden in diesem Abschnitt vorgestellt.

Informationsbeschaffung: Die Autoren bringen hervor, dass die Informationsbeschaffung sich als schwerer erwies als anfangs gedacht. Dies lag daran, dass Dokumentationen oft ungenügend waren oder viel zu einfache Anwendungsfälle beschrieben.

Adapter-Komponenten: Die Idee war die Nutzung von Adapter-Komponenten, die es in Swing und JavaFX gibt, um den Prozess der iterativen Migration zu beschleunigen. In Swing gibt es `JFXPanel`, in welches JavaFX-Komponenten eingelegt werden können, und für JavaFX gibt es `SwingNode`, in welche Swing-Komponenten eingebunden werden können. Die Nutzung dieser Adapter-Komponenten brachte jedoch das Problem der limitierten Dokumentation dieser Komponenten mit sich, denn diese zeigen nur einfache Szenarien, in denen diese Adapter verwendet werden können.

Leistungsprobleme: Bei der Nutzung von `SwingNode` kann es zu Leistungsproblemen kommen, weil diese nicht dafür gemacht sind, komplexe Swing-Komponenten zu halten.

Gleichnamige Klassen: Es kam zu falschen Annahmen über gleiche Funktionalitäten bei gleichnamigen Klassen, wie zum Beispiel bei der Klasse `Shape`, die in Swing und JavaFX gleich heißt, jedoch unterschiedliche Funktionalitäten aufweist. Diese falschen Annahmen führten zur Notwendigkeit eines Redesigns des originalen Designs, nachdem die Migration teilweise abgeschlossen wurde.

Funktionslücken: Die Migration des Zeichnens in Swing erwies sich als schwerer als anfangs gedacht, da das Zeichnen in JavaFX unterschiedlich funktioniert als in Swing. Daher war die Lösung nicht ganz simpel, weil man nicht einfach eine Klasse durch eine anderen ersetzen konnte.

Feature Blindheit: Für Features, die es in Swing nicht gab und deswegen selbst entwickelt wurden, wurde angenommen, dass es auch keine JavaFX-Lösung für diese Probleme gibt. Somit wurden für diese Features eigene Lösungen in JavaFX entwickelt. Zu einem späteren Zeitpunkt zeigte sich, dass es für diese Features in JavaFX eine vorgefertigte Komponente gibt.

2.7 Verwandte Arbeiten und Forschungsergebnisse

In diesem Abschnitt werden Arbeiten vorgestellt, die von hoher Relevanz für diese Arbeit sind.

Eine strukturierte Herangehensweise an ein Projekt wie die Softwaremigration stammt aus dem Buch mit dem Titel *Object-Oriented Reengineering Patterns* [11]. Dieses Buch liefert eine ausführliche Herangehensweise für ein besseres Verständnis des Projektausmaßes einer Softwaremigration. Es beschreibt ausführlich die Wichtigkeit des Verstehens der eigenen Legacy-Anwendung, um diese im nächsten Schritt gut migrieren zu können. Dieses Buch bildet eine Grundlage, um im nächsten Schritt eine geeignete Migrationsstrategie für die zu migrierende Anwendung finden zu können.

Eine Arbeit, die für die Auswahl der Migrationsstrategie eine wichtige Rolle spielt, ist ein Forschungsbericht mit dem Titel *Software Migration: A Theoretical Framework* [7]. Diese Arbeit gibt einen Überblick über einige technische (Black-, White- und Gray-Box) und zeitliche Migrationsstrategien (Big-Bang und inkrementell), die bei der Auswahl einer passenden Migrationsstrategie behilflich sind.

Die KI ist in dieser Arbeit ein wichtiges Tool zur Unterstützung bei der Modernisierung einer bestehenden Anwendung. Die Idee, dass die KI in vielen verschiedenen Bereichen der Softwareentwicklung verwendet werden könnte, stammt aus einem Blogpost des Unternehmens Explicatis [4], das praxisnahe Erfahrungen mit dem aktuellen Stand der KI innerhalb der Softwareentwicklung besitzt.

Eine verwandte Arbeit ist die Studie *Lessons Learned While Migrating From Swing to JavaFX* [22], in der die Autoren ihre Erfahrungen aus der Migration ihrer Anwendung von einem GUI-Framework auf ein anderes GUI-Framework dokumentieren und dabei wichtige Herausforderungen und Lösungen vorstellen. Die in der Studie genannten Herausforderungen wurden detailliert in Abschnitt 2.6 beschrieben. Eine in der Studie erwähnte Lösung für die Migration der GUI-Komponenten ist die Verwendung von `SwingNode` zur Integration der Swing-Komponenten in eine JavaFX-GUI.

3 Methodik

In diesem Kapitel steht die Methodik im Vordergrund. Zuerst wird die Herangehensweise bei der Analyse des Legacy-Codes beschrieben. Anschließend wird das Thema der Migrationsstrategie behandelt. Dabei werden verschiedene Ansätze vorgestellt, unter anderem auch die in dieser Arbeit gewählte Strategie. Nach der Vorstellung der Migrationsstrategie werden die Werkzeuge und Techniken präsentiert, die bei der Migration zum Einsatz kamen. Außerdem erfolgt eine Risikoanalyse, um potenzielle Probleme vor der Implementierung im nächsten Kapitel zu identifizieren und entsprechende Vorbereitungen zu treffen.

3.1 Analyse der bestehenden Anwendung

Die Analyse der Anwendung folgt dem in [11] beschriebenen Vorgehen. Vor einer Überlegung der Migration steht das Verständnis der Codebasis im Vordergrund. Dieses Verständnis bildet die Grundlage für eine erfolgreiche Migration. Hierzu können gegebenenfalls Unified Modeling Language (UML)-Diagramme, Architekturdiagramme oder Ähnliches entwickelt werden, um das Verständnis zu fördern. Dieser Abschnitt lässt sich auch als *Reverse Engineering* bezeichnen, da die Anwendung vor der Entwicklung einer Migrationsstrategie und der eigentlichen Migration analysiert wird. *Reverse Engineering* bedeutet die Analyse eines bestehenden Systems, mit dem Ziel ein besseres Verständnis über es zu erlangen [11, Kapitel 1.2]. Im Prozess der Analyse der bestehenden Anwendung treten Fragen auf, die geklärt werden müssen, und der Fokus wird auf das Problem gerichtet, das identifiziert und behoben werden soll. In diesem Fallbeispiel besteht das Problem in der veralteten GUI-Technologie, die durch eine neue Lösung modernisiert werden soll.

3.1.1 Planung der Analyse

Laut der Beschreibung in [11, Kapitel 2] ist es wichtig, eine Strategie für das Projekt zu entwickeln, um sicherstellen zu können, dass ein roter Faden aufgebaut und aufrechterhalten werden kann. Es gibt Prinzipien und Richtlinien, um diesen roten Faden aufrechterhalten zu können. Einige der im Rahmen dieser Migration angewendeten Prinzipien und Richtlinien umfassen *„Most Valuable First“* [11, Kapitel 2.4], bei dem es wichtig ist, zu verstehen, welche Probleme zuerst priorisiert werden sollten, und *„Keep it Simple“* [11, Kapitel 2.7], bei welchem einem mitgegeben wird, dass die einfache Lösung einer komplexeren vorgezogen werden sollte.

In diesem Fallbeispiel mit der veralteten GUI-Technologie wäre somit die Priorisierung der zu migrierenden GUI-Komponenten notwendig. Diese Priorisierung ist entscheidend, um festzulegen, welche Komponenten für die Migration von besonderer Bedeutung sind und zuerst migriert werden sollten. Danach gilt in der Migration, dass in manchen Fällen Kompromisse eingegangen werden sollten, da das Ziel nicht ist, gegen die neue Technologie zu arbeiten, sondern mit ihr, um die Lösung einfach zu gestalten und Komplexität zu vermeiden.

3.1.2 Beginn der Analyse

In erster Instanz findet eine Kommunikation mit den Beteiligten der entsprechenden Anwendung statt, um einen Überblick über den aktuellen Stand der Anwendung zu erhalten [11, Kapitel 3.1]. Anschließend wird die Dokumentation innerhalb eines festgelegten Zeitraums gelesen, und dabei werden Notizen erstellt, die relevante Informationen festhalten, die für das Reengineering nützlich sein könnten. Aufbauend auf diesen Erkenntnissen wird die Codebasis in einer Stunde gelesen [11, Kapitel 3.3]. Reengineering ist der Prozess der Verbesserung oder Modernisierung eines bestehenden Systems [11, Kapitel 1.2].

Abschließend wird in [11, Kapitel 3.5] beschrieben, dass in diesem Abschnitt der Analyse die Anwendung, sofern vorhanden, zum ersten Mal ausgeführt wird, um die Dokumentation und den Code besser nachvollziehen zu können. Nach diesem ersten Kontakt mit der Anwendung und der durchgeführten Analyse zeigt sich, ob eine Migration möglich ist oder nicht.

Als Fallbeispiel dient hier die Migration von Java Swing und Java AWT zu JavaFX. Der Fokus liegt daher auf den Klassen, die diese GUI-Technologien enthalten, was den

Umfang der zu betrachtenden Klassen bereits reduziert. Wie in der Planungsphase der Analyse erwähnt, soll das Wichtigste zuerst migriert werden. Um dies zu erreichen, werden in diesem Abschnitt die GUI-Komponenten nach ihrer Komplexität aufgelistet. Die Bewertung der Komplexität einer Klasse erfolgt in dieser Arbeit anhand der Anzahl ihrer Abhängigkeiten sowie der Zeilenanzahl des Codes.

3.1.3 Aufbau eines tieferen Verständnisses der Anwendung

Die in diesem Abschnitt genannten Vorgehensweisen stammen aus [11, Kapitel 4 und 5] und wurden direkt auf das Fallbeispiel angewendet und anhand dieser gezeigt.

In diesem Abschnitt besteht das Ziel darin, nach der Entscheidung über die Machbarkeit der Migration ein tieferes Verständnis der Anwendung aufzubauen. Im Fallbeispiel dieser Arbeit generiert und stellt die betrachtete Java-Anwendung UML-Diagramme dar, weshalb ein solides Verständnis von UML notwendig ist. Anschließend werden die für das Zeichnen der Diagramme relevanten Komponenten näher untersucht, insbesondere im Hinblick auf die eingesetzten Technologien.

Im Vergleich zum Beginn der Analyse ist es wichtig, die Dokumentation nicht nur erneut zu lesen, sondern auch die darin beschriebenen Tests nachzuvollziehen und zu verinnerlichen. Dadurch entsteht ein tieferes Verständnis der Abläufe und Funktionalitäten der Anwendung. Eine Möglichkeit zum Verständnis der GUI-relevanten Codeabschnitte ist das Durchlaufen der Anwendung im Debug-Modus. Dies geschieht mithilfe der Nutzungsszenarien, die während der Analyse der Dokumentation und Anwendung identifiziert wurden. Hier wird beobachtet, welche Objekte miteinander arbeiten und wie diese instanziiert werden. Dieser Vorgang ist sehr zeitaufwendig, weshalb er nur bei den Kernkomponenten des Fallbeispiels USE angewendet wird. Eine weitere Möglichkeit, um den Code und die Entwicklung der Anwendung besser zu verstehen, besteht darin, in die Versionshistorie zu schauen. Diese zeigt auf, welche Änderungen zwischen den Versionen vorgenommen wurden, und liefert Einblicke in die Gründe für diese Änderungen.

3.1.4 Risikoanalyse

Am Anfang einer Migration steht die Herausforderung, tiefgreifende Änderungen an einem Legacy-System vorzunehmen. Dabei müssen zahlreiche Risiken berücksichtigt werden. In dieser letzten Analysephase, in der ein gutes Verständnis des Legacy-Systems

vorliegt, geht es darum, potenzielle Risiken zu identifizieren, um diese im folgenden Kapitel der Implementierung minimieren zu können.

Um nach der Migration sicherzustellen, dass alle bestehenden Funktionen weiterhin korrekt arbeiten, ist eine umfassende Teststrategie essenziell. In vielen Legacy-Systemen sind jedoch entweder keine oder nur unzureichende Tests vorhanden [11, Kapitel Introduction].

In dem Fallbeispiel USE ist ein Problem, dass Zeitressourcen nicht effizient genutzt werden. Dies kann passieren, wenn der Fokus auf weniger relevanten Komponenten gelegt wird oder essenzielle Funktionen nicht vollständig oder zu spät integriert werden, weil die Priorisierung in 3.1.1 zu ungenau war. Darüber hinaus stellt die hohe Komplexität einzelner GUI-Komponenten ein Risiko dar. Manche Klassen besitzen eine hohe Kopplung oder sind sehr umfangreich und enthalten Tausende LOC, die sich möglicherweise nicht ohne Weiteres migrieren lassen oder den zeitlichen Rahmen der Arbeit sprengen. In solchen Fällen besteht die Gefahr, dass bestimmte Funktionen nur eingeschränkt übernommen werden können oder zusätzliche Umstellungen erforderlich werden.

3.2 Migrationsstrategie

Wie in [7, S. 1 f.] beschrieben, ist Softwaremodernisierung ein sich immer häufiger wiederholender Prozess. Da es sich bei der Softwaremodernisierung um ein Softwareentwicklungsprojekt handelt, ist sie anfällig für verschiedene Risiken und mögliche Fehlschläge. In diesem Abschnitt werden diverse Migrationsstrategien vorgestellt, um im folgenden Abschnitt eine Migrationsstrategie zu entwickeln, welche für das Fallbeispiel passend ist und Risiken sowie mögliche Fehlschläge minimiert.

3.2.1 Überblick über Migrationsstrategien

Eine Migration kann auf verschiedenste Art und Weise durchgeführt werden. Die Wahl der Strategie hängt von verschiedenen Faktoren ab, wie dem Umfang der Änderungen und der gewünschten Integrationstiefe. Im Folgenden werden Migrationsstrategien vorgestellt, die aus dem Forschungsbericht *Software Migration: A Theoretical Framework* [7, S. 11 ff.] stammen.

Die in der Literatur beschriebenen Migrationsstrategien lassen sich in zwei übergeordnete Kategorien einteilen: erstens in Strategien, die sich auf den technischen Zugriff auf das bestehende System beziehen, und zweitens in solche, die sich auf den zeitlichen Ablauf der Migration konzentrieren.

Technische Migrationsstrategien

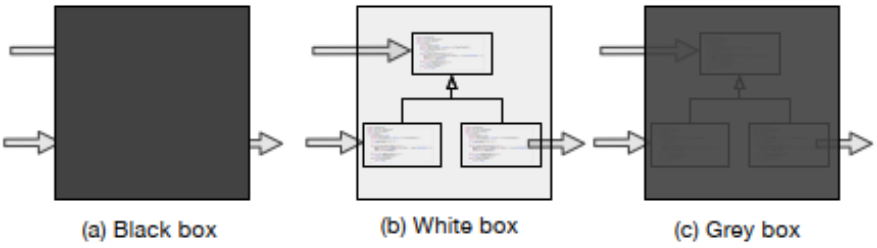


Abbildung 3.1: Ansätze[7, S. 12]

In Abbildung 3.1 werden die drei technischen Ansätze Black-Box, White-Box und Grey-Box vorgestellt. Diese Strategien repräsentieren die häufigsten Herangehensweisen zur Bewältigung von Reengineering-Herausforderungen.

Black-Box (Wrapping): Black-Box-Ansätze, die auch als externe Ansätze bekannt sind, ignorieren die interne Struktur des Systems. Stattdessen konzentrieren sie sich darauf, anhand der Eingaben und Ausgaben der Legacy-Anwendung deren Schnittstellen und Subsysteme besser zu verstehen (vgl. Abbildung 3.1 (a)). Diese Ansätze erfordern meist nur minimale oder gar keine Anpassungen am bestehenden System und stützen sich häufig auf Wrapping-Verfahren. Beim Wrapping wird die Software mit einer zusätzlichen Schicht umhüllt, die die vorhandene Komplexität versteckt und eine neue Schnittstelle bereitstellt. Dadurch lassen sich Inkompatibilitäten zwischen der bereitgestellten Schnittstelle und der von der aktuellen Integration benötigten Schnittstelle beheben, indem das Wrapping als Vermittler zwischen ihnen dient. Dies geschieht beispielsweise durch die Nutzung externer Lösungen wie Application Programming Interfaces (APIs). Wrapping erfordert oft die Entwicklung neuer Code-Komponenten, die das ursprüngliche System (Black Box) in die neue Umgebung integrieren.

White-Box (Transformation): White-Box-Ansätze analysieren die interne Struktur eines Systems und beruhen oft auf Reverse Engineering, um ein tiefgehendes Verständnis der Systemkomponenten und Subsystemkomponenten zu erlangen (vgl. Abbildung 3.1 (b)). Ziel ist es, Elemente und ihre Abhängigkeiten auf verschiedenen Abstraktionsebenen zu identifizieren, etwa Klassen, Muster oder Beziehungen. Diese Methode erfordert meist umfangreiche Änderungen am bestehenden System und setzt oft auf Transformationsverfahren. Die Transformation entwickelt eine semantisch äquivalente Softwarekomponente, die sich technologisch von der ursprünglichen unterscheidet, aber denselben Zweck erfüllt. Da diese Methode direkten oder indirekten Einfluss auf den Quellcode hat, kann sie auf verschiedenen Systemebenen angewendet werden, darunter Architektur, Design, Programmiersprachen, APIs, Paradigmen und externe Abhängigkeiten.

Grey-Box (Hybrider Ansatz): Grey-Box-Migration kombiniert Black-Box- und White-Box-Ansätze, weshalb diese auch ein hybrider Ansatz genannt wird (vgl. Abbildung 3.1 (c)). Interne Methoden werden verwendet, um eine detaillierte Anpassung zu ermöglichen, und gleichzeitig verwenden sie externe Methoden, um Risiken zu minimieren und den Aufwand für aufwendige interne Änderungen zu reduzieren. Es gibt somit zwei Hauptansätze für die Grey-Box-Migration: den internen Ansatz und den externen Ansatz. Im internen Ansatz wird das System in Teile zerlegt und analysiert. Dadurch können einzelne Teile des Systems in sogenannte Wrapper gekapselt werden, anstatt das gesamte System zu erfassen. Somit findet dieser Ansatz seinen Gebrauch meistens in der Migration von Software in eine Service-Architektur. Im externen Ansatz, welcher oft bei Modernisierungen vorkommt, werden Teile des Systems mithilfe von externen Lösungen verwaltet. Wie in einem Beispiel einer Migration von der Datenverwaltung in COBOL, die direkt im Code geschah und später in RDBM überführt wurde.

Zeitliche Migrationsstrategien

Neben dem technischen Zugriff spielt auch der Ablauf der Migration eine wichtige Rolle. Zwei bekannte Ablaufstrategien sind die Big-Bang-Migration und die inkrementelle Migration.

Big-Bang-Migration: Die Beschreibung der Big-Bang-Migration folgt dem in [8, S. 1147 f.] dargestellten Vorgehen. Die Migration der gesamten Anwendung erfolgt in einem Schritt. Daher wird diese Migrationsstrategie als risikoreich eingestuft, weil es sehr schwer ist, die Bedingungen einer produktiven Umgebung vollständig neu zu erstellen. Da die

Umgebung sehr viele Systemabhängigkeiten besitzt, kann der Fehler in einem Element zu Problemen in anderen Modulen führen. Die Folge davon ist, dass die Wiederherstellung im schlimmsten Fall fatal oder sehr schwer ist.

Inkrementelle Migration: Der Prozess der inkrementellen Migration folgt der Herangehensweise, die Migration schrittweise zu verwirklichen. Daher werden Features und Komponenten nacheinander migriert [8, S. 1150].

Der Vorgang der Migration einer Legacy-Anwendung ist ein riskantes Vorgehen, das viele Probleme mit sich bringt. Das Ziel ist es, diese zu minimieren. Um das Risiko in der Migration zu minimieren, wird am häufigsten ein inkrementeller Ansatz gewählt. [7, S. 13]

3.2.2 Auswahl der Migrationsstrategie

Die Auswahl der Migrationsstrategie in diesem Projekt orientiert sich an einer Kombination aus bekannten Strategien, wie sie unter anderem in [7] und in der Beschreibung des Strangler-Fig-Patterns nach Martin Fowler [12] vorgestellt werden.

Für die Modernisierung des Legacy-Systems wurde eine inkrementelle Migration in Kombination mit einer White-Box- und Grey-Box-Strategie gewählt.

Die inkrementelle Migration in diesem Projekt basiert auf dem Prinzip der Strangler-Fig-Strategie, die Martin Fowler als eine gute Methode zur schrittweisen Ablösung von Altsystemen beschreibt [12].

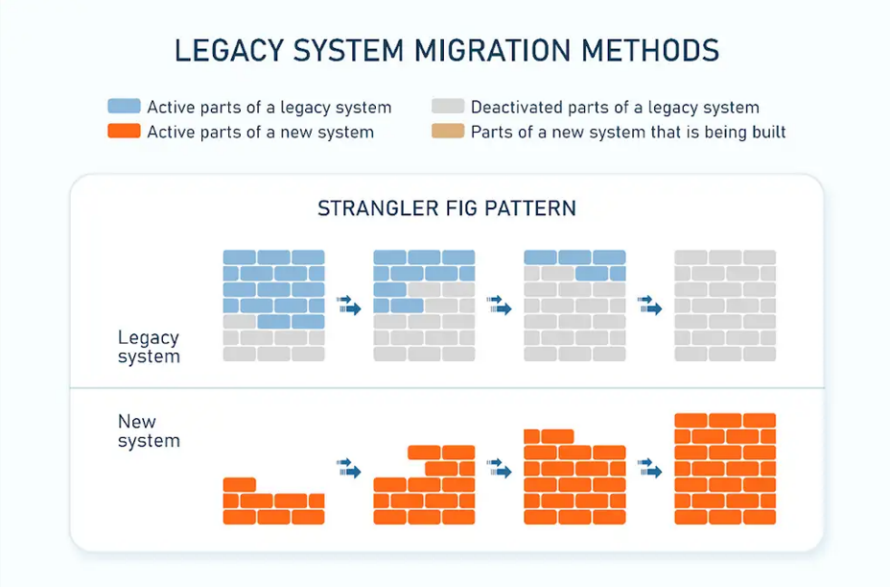


Abbildung 3.2: Strangler Fig Pattern[24]

Laut [12] handelt es sich bei der Strangler-Fig-Strategie um eine schrittweise Modernisierung. Dieser beginnt mit kleinen, oft neuen Funktionen, die parallel zur bestehenden Codebasis entwickelt werden. Zur gleichen Zeit werden schrittweise Teile des Verhaltens des Altsystems in den neuen Code übertragen, bis das alte System schließlich vollständig ersetzt ist.

In diesem Projekt bedeutet dies die schrittweise Migration der GUI-Komponenten von Swing/AWT in JavaFX. Um diese parallele Entwicklung zu schaffen, werden die GUI-Komponenten nacheinander migriert wie auf (vgl. Abbildung 3.2) zu sehen ist. Für die Klassen wird jeweils eine neue Klasse erstellt, welche die JavaFX-Migration enthalten wird. Wenn komplett migriert wurde, kann der alte Code durch den neuen ersetzt werden.

Die Idee von Grey-Box-Strategien ermöglicht es, in dieser ausgewählten Migrationsstrategie komplexe Swing-Komponenten, bei denen eine vollständige Migration wegen zeitlicher Limitation nicht möglich ist, temporär in die JavaFX-Lösung einzubinden. Dadurch bleibt die vorhandene Funktionalität erhalten, während neue Komponenten nach und nach in JavaFX implementiert werden. Das langfristige Ziel ist jedoch die komplette Migration des GUIs der Legacy-Anwendung von Swing/AWT auf JavaFX, so wie es aus White-Box-Strategien bekannt ist.

Ein großer Vorteil der Strangler-Fig Migrationsstrategie ist das geringe Risiko, welches durch die kontrollierte schrittweise Migration erreicht wird, welche gegenüber einer Big-Bang-Umstellung weniger fehleranfällig ist.

Nach [7, S. 12] liegt ein Vorteil hybrider Lösungen in der Reduzierung von Risiken sowie in der Verringerung vom Aufwand, indem sie eine Verbindung zwischen internen und externen Methoden schaffen.

Ein weiterer Vorteil laut [12], welcher diese Migrationsstrategie kennzeichnet, ist die frühzeitige Wertschöpfung von neuen Komponenten, da diese bereits genutzt werden können, bevor die vollständige Migration abgeschlossen ist.

3.3 Werkzeuge und Techniken

In diesem letzten Abschnitt des Kapitels werden die Werkzeuge und Techniken vorgestellt, die während der Migration der GUI einer bestehenden Anwendung eingesetzt wurden.

Nutzung vorhandener Dokumentation: Ein wichtiges Werkzeug für die erfolgreiche Migration einer Legacy-Anwendung ist die vorhandene Dokumentation. Zur Analyse und zum tieferen Verständnis der bestehenden Softwarearchitektur wurden Diagramme, Beispiele und Erklärungen betrachtet, die in der Dokumentation der zu migrierenden Anwendung enthalten sind. Diese ermöglichen ein schnelles Verständnis der Komponenten, ihrer Abhängigkeiten sowie der Funktionsweise der Anwendung. Damit bildet dies eine wichtige Grundlage für die Planung und Durchführung der Migration [26].

Risikominimierung: Um das Risiko während der Migration möglichst gering zu halten, wurde der von [7, S. 14] beschriebene Ansatz verfolgt: „Minimizing the migration risk is a key requirement. The most common strategy is an incremental approach to minimise the risk“ [7, S. 14]. Dementsprechend wurde eine inkrementelle Vorgehensweise gewählt, bei der zunächst die Grundlagen der bestehenden Anwendung detailliert analysiert und verstanden wurden. Die anschließende Migration erfolgte schrittweise nach dem Strangler-Fig-Prinzip, wie im vorherigen Abschnitt beschrieben. Auf diese Weise konnte das Risiko unerwarteter Probleme reduziert werden.

Vergleichendes Testverfahren zur Qualitätssicherung: Da in dieser Anwendung zum Zeitpunkt des Projektes keine GUI-Tests existieren und automatisierte GUI-Tests aufgrund der Komplexität im Rahmen dieser Arbeit nicht implementiert werden konnten, erfolgt die Validierung der GUI anhand eines vergleichenden Testverfahrens, welches sich am Konzept eines Regressionstests orientiert. Bei einem Regressionstest werden bestehende Funktionalitäten einer Anwendung systematisch geprüft, um nach Änderungen (wie einer Migration) sicherzustellen, dass keine unbeabsichtigten Seiteneffekte auftreten [23, S. 70 f.]. In diesem Fall wurden vorhandene Funktionen des Altsystems mit denen des migrierten Systems verglichen, um Unterschiede und Fehler gezielt zu identifizieren.

IntelliJ IDEA als Entwicklungsumgebung: Als Entwicklungsumgebung wurde IntelliJ IDEA verwendet, welche effizientes Arbeiten durch leistungsfähige Funktionen wie Debugging, Refactoring und Code-Analyse unterstützt. Insbesondere bei größeren, komplexeren Anwendungen, wie im Falle dieser Migration, bietet IntelliJ IDEA eine übersichtliche Navigation und erleichtert somit den Migrationsprozess. Darüber hinaus wurde die in IntelliJ IDEA integrierte Diagram-Funktion verwendet [15]. Diese Funktion ermöglicht es, Klassen inklusive ihrer Methoden, Attribute und Abhängigkeiten visuell in Diagrammen darzustellen. Dadurch lassen sich Zusammenhänge zwischen den einzelnen Klassen gut erkennen. Dies erlaubt es, Abhängigkeiten gezielt zu identifizieren und einzuschränken, ob die aktuell betrachtete Klasse komplex oder einfach zu migrieren ist. Die Diagramm-Funktion unterstützte so sowohl die Kategorisierung und Priorisierung der zu migrierenden Klassen als auch das tiefere Verständnis der Gesamtstruktur der Anwendung.

KI-unterstützte Migration mit ChatGPT: Im Rahmen dieser Arbeit wurde ChatGPT eingesetzt, um die Softwaremodernisierung von der Planung bis hin zur Implementierung zu unterstützen. Die Vorgehensweise und die Rolle der KI werden detailliert in Abschnitt 4.3 beschrieben.

Verwendung von FXML und Scene Builder für die GUI-Implementierung: Die Informationen zu FXML und dem Scene Builder stammen aus JavaBeginners.de [14] und bildeten die Grundlage für deren Einsatz in diesem Projekt. Für die Implementierung der neuen JavaFX-GUI wurde die XML-basierte Beschreibungssprache FXML verwendet. FXML dient dazu, den Objektbaum von JavaFX-basierten Benutzeroberflächen eindeutig zu spezifizieren, und ermöglicht eine klare Trennung zwischen Benutzeroberfläche

(View) und Anwendungslogik (Controller) gemäß dem Model-View-Controller (MVC)-Entwurfsmuster. Dadurch wird die Gestaltung der grafischen Oberfläche weitgehend vom Rest des Programmentwurfs entkoppelt, was Wartbarkeit und Erweiterbarkeit verbessert. Zur initialen Gestaltung der Benutzeroberfläche wurde anfangs, wie von JavaBeginners.de [14] vorgeschlagen, der visuelle Editor JavaFX Scene Builder verwendet, um schnell eine visuelle Vorlage der Oberfläche zu erstellen. Anschließend erfolgte die Finalisierung und Anpassung der Oberfläche direkt im generierten FXML-Code. Der Scene Builder erwies sich besonders bei der Positionierung und Strukturierung komplexer GUI-Komponenten als hilfreiches Werkzeug.

4 Implementierung

Dieses Kapitel zeigt die Anwendung der im vorherigen Kapitel vorgestellten Migrationsstrategie. Im Mittelpunkt steht die schrittweise KI-unterstütze Migration der GUI-Komponenten in die Zielsprache. Im Folgenden werden die durchgeführten Migrationsschritte und die Rolle der KI im Migrationsprozess vorgestellt.

4.1 Anpassung der Softwarearchitektur

Java entwickelt sich immer weiter, wodurch sich im Laufe der Zeit vieles verändert. Aufgrund der ausgewählten Migrationsstrategie wird die GUI von USE inkrementell migriert, angelehnt an das Prinzip der im vorherigen Kapitel vorgestellten Strangler-Fig-Strategie. Die GUI-Komponenten werden schrittweise migriert, um die Funktionalität der Legacy-Anwendung zu sichern und parallel die neue JavaFX-Anwendung zu integrieren. Dies ermöglicht es, das Legacy-System in einer zukünftigen Arbeit vollständig abzulösen. Um JavaFX einbauen zu können, sind jedoch folgende Anpassungen an der bestehenden Softwarearchitektur notwendig.

4.1.1 Technologische Rahmenbedingungen

Seit Java 11 ist JavaFX nicht mehr Bestandteil des offiziellen Oracle Java Development Kits (JDKs), sondern wird unter dem Namen OpenJFX als eigenständiges Projekt bereitgestellt [18]. Damit JavaFX in diesem Projekt eingesetzt werden kann, muss das JavaFX Software Development Kit (SDK) zusätzlich zum verwendeten JDK im Projekt eingebunden werden.

4.1.2 Modulsystem und Kompatibilität

Wie in [13] dargestellt, wurde mit Java 9 ein Modulkonzept eingeführt, was eine weitere Gliederung oberhalb der Package-Struktur erlaubt. Dieses Konzept verbessert unter anderem die Sicherheit der Steuerung von Zugriffsrechten und die Skalierbarkeit der Anwendungen.

Die folgenden Informationen stammen aus der offiziellen OpenJFX-Dokumentation [21]. Da JavaFX, wie im vorherigen Abschnitt erläutert, nicht mehr Teil des JDK ist und als externe Bibliothek eingebunden werden muss, ist in modularen Anwendungen eine explizite Angabe der JavaFX-Module erforderlich. Diese erfolgt in der Datei `module-info.java` über `requires`. Wird FXML verwendet, ist zu beachten, dass JavaFX mit Reflektion auf den Controller zugreift. Aus diesem Grund muss das entsprechende Paket mit `opens` für das Modul `javafx.fxml` geöffnet werden. Um gezielt Klassen für andere Module verfügbar zu machen, können Pakete mit `exports` veröffentlicht werden. Im Folgenden ist ein Beispiel für solch eine `module-info.java`-Datei:

Listing 4.1: Beispiel `module-info.java`, Quelle: [21]

```
module hellofx {
    requires javafx.controls;
    requires javafx.fxml;
    opens org.openjfx to javafx.fxml;
    exports org.openjfx;
}
```

Fehlt etwas in der `module-info.java`-Datei, kommt es beim Kompilieren oder zur Laufzeit zu Fehlern. Beispielsweise wegen nicht auflösbarer Module oder fehlender Zugriffsrechte auf interne Klassenstrukturen.

4.1.3 Auswirkungen auf die bestehende Architektur

Bevor die eigentliche Migration beginnen konnte, mussten zunächst die erwähnten notwendigen architektonischen Anpassungen vorgenommen werden.

Das migrierte Projekt besteht aus den folgenden drei Modulen, in denen jeweils spezifische Anpassungen für die Integration von JavaFX vorgenommen wurden:

use-assembly bündelt die gesamte Anwendung, verwaltet Plugins und enthält Build-Anweisungen wie die `assembly.xml`-Datei.

Listing 4.2: fileSet-Anweisung für die Einbindung des JavaFX-SDK

```
<fileSet>
    <directory>${project.basedir}/../use-gui/lib/javafx-sdk-21.0.5</
        directory>
    <outputDirectory>lib/javafx-sdk-21.0.5</outputDirectory>
    <includes>
        <include>lib/**</include>
        <include>bin/**</include>
        <include>legal/*
        </include>
     </includes>
</fileSet>
```

Innerhalb von `assembly.xml` wurde eine zusätzliche fileSet-Anweisung eingebaut, wie in Listing 4.2 zu sehen ist. Diese Anweisungen sind Maven-spezifische Definitionen, die in der `assembly.xml`-Datei für die Angabe von Dateikopiervorgängen verwendet werden. Die neue fileSet-Anweisung wurde eingebaut, um das benötigte JavaFX-SDK aus dem `lib`-Verzeichnis des Moduls `use-gui` in das Ausgabepaket zu übernehmen und so die Ausführbarkeit der modularen JavaFX-Anwendung sicherzustellen.

use-core enthält die gesamte Anwendungslogik sowie das interne Modell. In diesem Modul wurde eine `module-info.java`-Datei ergänzt, um die Kompatibilität mit dem integrierten Java-Modulsystem herzustellen. Weitere strukturelle Änderungen waren nicht erforderlich.

use-gui stellt die grafische Benutzeroberfläche auf Basis von Swing und AWT bereit. Zur Verwendung von JavaFX wurde das SDK heruntergeladen und lokal im Projekt im Verzeichnis `use-gui/lib` abgelegt. Dieses SDK wird nicht über Maven geladen, sondern manuell eingebunden, um die Kompatibilität zur modularisierten Struktur sicherzustellen. Beim Bauen der Anwendung mit Maven sorgt eine spezielle fileSet-Anweisung in der `assembly.xml` von `use-assembly` dafür, dass das JavaFX-SDK in das Ausgabepaket aufgenommen wird.

Zur Integration von JavaFX wurde die ChatGPT gezielt nach einer geeigneten Struktur gefragt, um die neue GUI parallel zur bestehenden Swing-Oberfläche einzubinden. Eine der vorgeschlagenen Lösungen war, die JavaFX-Komponenten in separaten Verzeichnissen mit dem Zusatz *FX* unterzubringen. Für ein Verzeichnis namens `util` wurde

ein Verzeichnis namens `utilFX` erstellt, und so weiter. Diese Herangehensweise wurde übernommen, da sie eine klare Trennung der neuen Implementierung ermöglicht und gleichzeitig sicherstellt, dass die bestehenden Legacy-Komponenten unverändert bleiben können, sodass die Legacy-Anwendung weiterhin läuft. Darüber hinaus unterstützt dieser Aufbau die inkrementelle Migration nach dem Prinzip der Strangler Fig, da neue Funktionalitäten schrittweise ersetzt werden können, ohne dass das bestehende System sofort vollständig abgelöst werden muss.

Die Anwendung verfügt nun über zwei Einstiegspunkte. Standardmäßig wird die Swing-Oberfläche geladen. Wird beim Programmstart der Parameter *-jfx* an erster Stelle übergeben, startet stattdessen die JavaFX-Version. Diese Struktur ermöglicht es, beide GUIs parallel im Projekt zu führen, zu testen und schrittweise zu migrieren, ohne die Funktionsfähigkeit der bestehenden Anwendung zu beeinträchtigen.

Zudem wurde auch in diesem Modul eine `module-info.java` erstellt, um die für JavaFX benötigten Module explizit zu deklarieren. Damit wird eine saubere Integration im Sinne des Java-Module-Systems gewährleistet.

Sowohl `use-core` als auch `use-gui` enthalten damit jeweils eine `module-info.java`-Datei, um die Modularität und JavaFX-Kompatibilität sicherzustellen.

Innerhalb der `pom.xml`-Datei werden folgende im Projekt verwendeten Dependencies, wie `javafx-fxml`, `javafx-controls` und weitere hinzugefügt, darunter auch das `javafx-maven-plugin` [21].

4.2 Entwicklung der neuen GUI

Aufbauend auf den in Abschnitt 3.3 beschriebenen Grundlagen zu FXML und Scene Builder, wird in diesem Abschnitt die Umsetzung des neuen GUI-Layouts erläutert.

4.2.1 Visuelles Designkonzept

Zu Beginn wurde ChatGPT nach geeigneten Tools und Möglichkeiten zur Umsetzung des neuen GUI-Layouts befragt. ChatGPT lieferte eine Liste potenzieller Werkzeuge und Ansätze, wobei FXML in Kombination mit dem Scene Builder als besonders geeignet für

dieses Projekt erschien. Ausschlaggebend für die Wahl des Scene Builders war unter anderem dessen einwandfreie Integration in die verwendete Entwicklungsumgebung IntelliJ IDEA [16].

Bei der Gestaltung wurde darauf geachtet, dass alles was in der GUI der Legacy-Anwendung existiert, auch in der neuen GUI wiederzufinden ist. Ziel war es, dem Benutzer eine vertraute Umgebung zu bieten und die Bedienbarkeit nicht zu erschweren. Das identische GUI-Layout stellt sicher, dass Benutzer sich auch in der neuen GUI der Anwendung schnell zurechtfinden.

4.2.2 Layout mit FXML

Mit dem Scene Builder wurde zuerst ein grobes Layout für die neue GUI entworfen, das sich stark am Aufbau der Legacy-Anwendung orientiert. Der Scene Builder generierte auf Basis dieses Entwurfs eine `.fxml`-Datei, die im weiteren Verlauf manuell weiterentwickelt wurde, um JavaFX-spezifische Anforderungen besser berücksichtigen zu können.

Innerhalb der FXML-Datei wird über das Attribut `fx:controller="Pfad-zur-Controller-Klasse"` die Controller-Klasse festgelegt, welche die Anwendungslogik enthält. Zentrale UI-Elemente wie `TreeView`, `TextArea` oder `StatusBar` erhalten ein eindeutiges `fx:id="id"`, um sie im Controller direkt referenzieren zu können.

4.2.3 Anbindung der Controller

Listing 4.3: Beispiel Controller-Klasse

```
public class MainWindow implements Initializable {
    @FXML
    private TextArea fLogTextArea;

    public void initialize(URL url, ResourceBundle resourceBundle) {
        fLogPanel = new LogPanel(fLogTextArea);
        ...
    }

}
```

Die in der FXML-Datei definierten UI-Elemente werden im Controller über `@FXML`-annotierte Variablen gebunden. In Listing 4.3 ist die Anbindung des Textfeldes für das

UI-Element *Log* zu sehen, welches das *Ausgabefeld (6)* in Abbildung 2.1 darstellt. Mithilfe dieser Anbindung kann innerhalb der Controller-Klasse gezielt auf Interaktionen mit dem UI-Element reagiert werden. Unter anderem bei Klicks auf Menüeinträge oder beim Laden von Dateien. Zudem implementiert der Controller das Interface Initializable, wodurch sich einmalige Initialisierungen in der Methode initialize() durchführen lassen [21]. Dazu gehören zum Beispiel das Setzen von Listenern oder das Befüllen grafischer Komponenten wie `TreeView`, `MenuBar` oder `ToolBar` beim Start der Anwendung. Die Initialisierung anhand des Beispiels mit Ausgabefeld (Log) ist in Listing 4.3 zu sehen.

Wie im Abschnitt 3.3 beschrieben, orientiert sich der Einsatz von FXML am Model-View-Controller (MVC)-Prinzip. Durch das Auslagerung des GUI-Layouts in die FXML-Datei und der Anwendungslogik in die Controller-Klasse wird diese Trennung von View und Logik realisiert.

4.3 Rolle der KI im Migrationsprozess

Im Rahmen dieses Projekts wurde ChatGPT an zahlreichen Stellen als unterstützendes Werkzeug eingesetzt. Seine Verwendung erfolgte für die technischen Analyse und auch zur praktischen Umsetzung im Migrationsprozess.

Zu Beginn des Projekts wurde ChatGPT verwendet, um Informationen zu vergleichbaren Arbeiten und wissenschaftlichen Quellen zu erhalten. Eine hilfreiche Studie konnte gefunden werden. Aus dieser Studie [22] stammt der Ansatz in dieser Arbeit, komplexere oder stark gekoppelte Swing-Komponenten zunächst über die Adapter-Komponente `SwingNode` in die neue JavaFX-Struktur zu integrieren. Dieser Ansatz erleichterte und beschleunigte die schrittweise Migration deutlich. Die Verwendung von `SwingNode` wird detaillierter in Abschnitt 4.6 beschrieben.

In den frühen Projektphasen wurde ChatGPT unter anderem dazu genutzt, komplexe Codeabschnitte aus der Legacy-Anwendung zu analysieren und besser zu verstehen. Somit diente ChatGPT als Hilfstool im Prozess des *Reverse Engineering* des Projektes.

ChatGPT wurde hauptsächlich für die Migration der GUI-Komponenten eingesetzt. Für diesen Zweck wurden zwei eigene Vorgehensweisen entwickelt und wie folgt bezeichnet:

Top-Down: In diesem Ansatz wird eine vollständige Klasse inklusive konkreter Anweisungen zur Migration in das Modell eingegeben. Mit dieser Anweisung soll ChatGPT den

zu migrierenden Code analysieren und, wenn es in der Lage ist, eine brauchbare JavaFX-Version generieren. Diese Strategie wird zuerst eingesetzt. Untersucht wird auch, wie stark die Ergebnisqualität von der Komplexität oder Größe der Klasse abhängt.

Bottom-Up: Bei komplexen Klassen mit hoher Kopplung wird ein schrittweises Vorgehen gewählt. Hierbei werden einzelne Methoden nacheinander in das Modell gegeben. Diese werden zunächst analysiert und anschließend in JavaFX migriert. Ziel dieses kleinteiligen Vorgehens ist es, ein viel besseres Verständnis der ursprünglichen Struktur zu gewinnen und eine zuverlässigere Umsetzung zu erreichen.

Zusätzlich wurde ChatGPT während der gesamten Entwicklung in vielen verschiedenen wegen verwendet. Zum einen zur Generierung von Code, zur Beantwortung technischer Fragen, zur Formulierung von Kommentaren sowie zur Unterstützung bei der Fehlersuche.

Im folgenden Abschnitt wird darauf eingegangen, wie die GUI-Komponenten migriert und in das neue JavaFX-basierte System überführt wurden.

4.4 Migrationsprozess der GUI-Komponenten

In diesem Abschnitt geht es um die praktische Umsetzung der Migrationsstrategie, die im letzten Kapitel unter 3.2.2 vorgestellt wurde. Ziel ist es, eine JavaFX-Anwendung parallel zur existierenden Legacy-Anwendung zu integrieren.

4.4.1 Vorgehensweise

Die Migration erfolgte komponentenweise nach dem Prinzip der inkrementellen Migration. Hierbei kamen *White-Box-* und *Grey-Box-Strategien* zum Einsatz, wie vorher beschrieben. Die Umsetzung der Migration orientierte sich an der *Strangler-Fig-Strategie*, welche die schrittweise Ablösung des Altsystems ermöglicht. Eine Priorisierung der zu migrierenden Komponenten wurde nach dem im vorherigen Kapitel beschriebenen Verfahren erstellt. Wie schon beschrieben richtet sich die Priorisierung der Komponenten danach, wie einfach sie zu migrieren sind. Der Schwierigkeitsgrad wurde hier anhand der Komplexität und der Kopplung der Komponenten bestimmt. In diesem Projekt werden zu den bestehenden Swing-Komponenten die gleich funktionierenden JavaFX-Komponenten entwickelt, um die parallele Existenz zu ermöglichen.

4.4.2 Migration

Im letzten Abschnitt wurde ein GUI-Layout mithilfe von FXML und Scene Builder erstellt. Für diese GUI-Komponenten wurden in diesem Abschnitt die dazugehörenden Funktionalitäten in JavaFX implementiert.

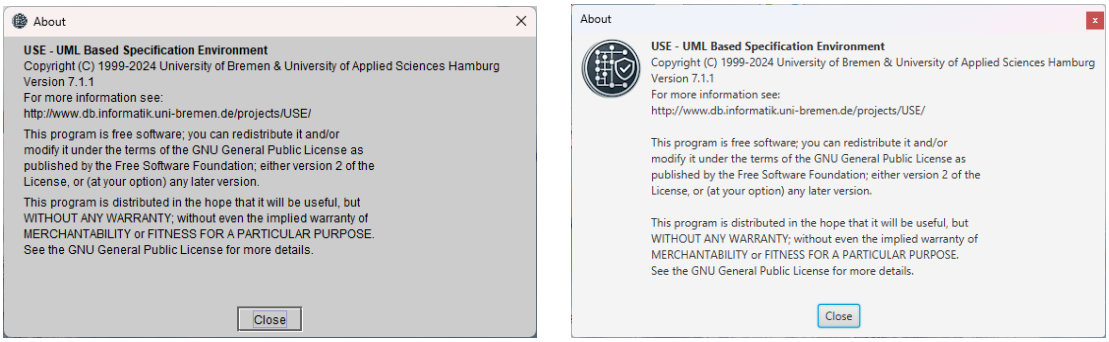
Die Migration beginnt mit den am einfachsten zu migrierenden Komponenten. Somit können die Komponenten nacheinander und unabhängig voneinander migriert und getestet werden. Im folgenden Abschnitt 4.5 wird näher auf das verwendete Testverfahren eingegangen sowie auf die Gründe, warum dieses gewählt wurde. ChatGPT wird als unterstützendes Tool in der Migration der Klassen verwendet. Es wird nach dem Vorgang, wie in Abschnitt 4.3 beschrieben eingesetzt. Zu Beginn wird die Migration mithilfe des Top-Down-Ansatzes durchgeführt, und falls diese scheitert, wird es mit dem Bottom-Up-Ansatz versucht. Um diese beiden Fälle zu veranschaulichen werden drei verschiedene Komponenten, die migriert wurden, vorgestellt.

Bei der ersten Klasse handelt es sich um die Klasse `AboutDialog`, die dafür zuständig ist, beim Klick auf die Menükomponente einen Dialog mit Informationen zur Anwendung anzuzeigen. Dieser Dialog enthält Versionsdetails, Lizenzhinweise und einen Button zum Schließen des Fensters. Die Klasse enthält ungefähr 110 Lines Of Code (LOC), ist somit sehr klein und nicht komplex. Es findet keine Datenverarbeitung und auch keine Interaktion mit anderen Komponenten statt. In dieser Klasse gibt es keine externen Abhängigkeiten außer GUI-Komponenten. Die Abbildung 4.1 zeigt den einfachen Aufbau der Klasse *AboutDialog*.

Listing 4.4: Prompt an ChatGPT (Top-Down-Ansatz)

```
Die folgende Klasse von Swing/AWT zu JavaFX migrieren: [CODE]
```

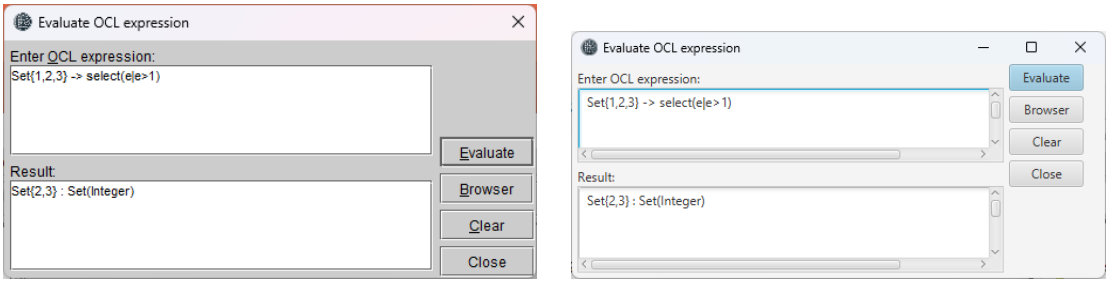
Als Prompt wurde ChatGPT eine einfache Anweisung, gefolgt von der vollständigen Klasse übergeben, wie in Listing 4.4 zu sehen ist. Als Ergebnis wurde ein vollständig funktionierender AboutDialog in JavaFX generiert. Diese Migration fiel ChatGPT leicht und erleichterte die Aufgabe, da die Klasse klein und kompakt ist und zu den verwendeten Imports jeweils eine äquivalente JavaFX-Lösung existiert. In diesem Fall konnte die Klasse vollständig übernommen werden, da ChatGPT eine Klasse generierte, die denselben Zweck erfüllte wie die Swing-Version, außerdem gut strukturiert ist und bereits Kommentare enthielt.



(a) Swing-Version (b) JavaFX-Version

Abbildung 4.1: AboutDialog vor und nach der Migration.

In der nächsten Klasse handelt es sich um eine große und etwas komplexere Klasse. Sie nennt sich `Evaluate OCL Expression` und ist für die Auswertung von Object Constraint Language (OCL)-Ausdrücken zuständig. Das Dialogfenster enthält ein Eingabefeld für die OCL-Ausdrücke und ein Ausgabefeld für die Ergebnisse der Auswertungen. Außerdem enthält es vier Buttons: einen Evaluate für die Evaluation der Ausdrücke, einen Browser zum Öffnen eines Browsers für detailliertere Anzeigen der Evaluation, einen Clear für das Leeren des Ausgabefeldes und einen Close für das Schließen des Evaluate OCL Expression-Fensters. Die Felder und Buttons sind in der Abbildung 4.2 gut zu erkennen. Die Klasse enthält ungefähr 319 LOC und ist etwas größer als die vorherige Klasse. Diese Klasse hat im Vergleich zur vorherigen eine etwas höhere Kopplung, da sie von USE-spezifischen Klassen Gebrauch macht, eine Datenverarbeitung und Interaktionen mit anderen Klassen stattfinden. Als Prompt wurde ChatGPT dieselbe Anweisung wie in der vorherigen Klasse mitgegeben 4.4. Als Ergebnis entstand eine vollständige Migration der Klasse, welche jedoch nicht ohne Weiteres übernommen werden konnte, da diese nicht einwandfrei funktionierte. Dieses Problem entstand aufgrund der Halluzinationen von ChatGPT, da es über die Funktion der verwendeten Komponenten von außerhalb spekulierte. Somit kam es zu diversen Fehlern wie fehlenden Funktionen und unvollständigen Lösungen. Um dieses Problem in der unvollständig generierten Lösung zu beheben, mussten einzelne Bereiche der Lösung manuell oder schrittweise mithilfe von ChatGPT nach dem Prinzip des Bottom-Up-Vorgehens überarbeitet werden.



(a) Swing-Version (b) JavaFX-Version

Abbildung 4.2: Evaluate OCL expression vor und nach der Migration

In dieser letzten, noch größeren und komplexeren Klasse, dem `ExprEvalBrowser`, der beim Betätigen des Browser-Buttons von der vorherigen Klasse geöffnet wird, werden die Auswertungen von OCL-Ausdrücken in einer Baumstruktur dargestellt. In diesem Browser können diverse weitere Filterungen und Highlightings vorgenommen werden, sowie zwei weitere Fenster aufgeklappt werden, die weitere Informationen wie Teilausdrücke und Variableninitialisierungen anzeigen. Die genannten Felder und Filterungen sind in der Abbildung 4.3 gut zu erkennen. Diese Klasse besteht aus über 1000 LOC und arbeitet sehr stark mit AWT- und Swing-Komponenten. Bei der Verwendung desselben Prompts 4.4 als Anweisung für ChatGPT wie in den letzten beiden Klassen scheiterte die Migration mit dem Top-Down-Ansatz. ChatGPT lieferte bei großen Klassen lediglich eine Art schrittweises Vorgehen und Beispielmigrationen der einzelnen Bereiche innerhalb der Klasse. Daher wurde bei diesen Klassen stattdessen der Bottom-Up-Ansatz gewählt, mit dem die Methoden innerhalb der Klasse schrittweise migriert werden konnten. Zu Beginn der Konversation wurde ChatGPT die komplette Klasse mitgegeben, damit ChatGPT den Kontext und die Funktion der einzelnen Methoden, die nacheinander geschickt und migriert werden sollen, besser nachvollziehen konnte.

Listing 4.5: Einleitender Prompt an ChatGPT (Bottom-Up-Ansatz)

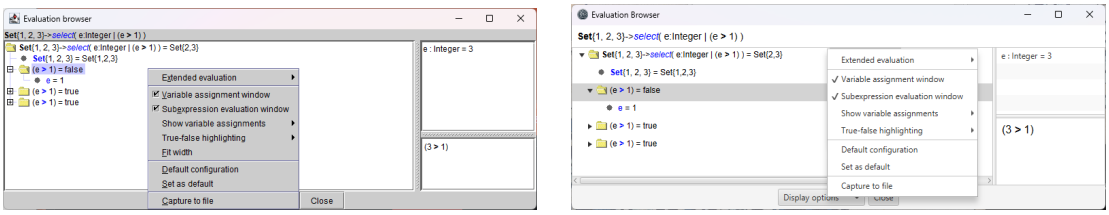
```
Ich brauche in dieser Konversation deine Hilfe bei der Migration von
    einzelnen Methoden der folgenden Klasse: [Code]
```

Folgend des einleitenden Prompts wird schrittweise für die Migration der Methoden folgendes Prompt verwendet:

Listing 4.6: Hauptprompt an ChatGPT (Bottom-Up-Ansatz)

```
Die folgende Methode soll von Swing/AWT zu JavaFX migriert werden: [CODE]
```

Mit dem Bottom-Up Ansatz war es ChatGPT möglich präziser auf einzelne Methoden einzugehen und auch genauere, vollständigere Lösungen zu generieren.



(a) Swing-Version (b) JavaFX-Version

Abbildung 4.3: Evaluation Browser vor und nach der Migration.

Nach diesem obigen Vorgehen wurden alle GUI-Komponenten migriert, bis auf sehr komplexe UML-Komponenten wie die Erstellung von Klassendiagrammen, Objektdiagrammen und weiteren, die den zeitlichen Rahmen dieser Arbeit sprengen würden. Für die Vollständigkeit der Funktionalitäten wurden diese Komponenten nach dem Prinzip eines Grey-Box-Ansatzes migriert. Auf diesen Ansatz wird in Abschnitt 4.6 detaillierter eingegangen.

4.5 Testverfahren

Die Migration der Komponenten ist sehr wichtig. Genauso wichtig ist auch das Testen des Ergebnisses, um sicherstellen zu können, dass die neue JavaFX-Komponente die gleichen Funktionalitäten besitzt wie die Legacy-Komponente. Um das sicherzustellen, wurde das zuvor in Abschnitt 4.5 vorgestellte Testverfahren angewendet. Das Testen nach dem vergleichenden Testverfahren findet nach jeder migrierten Komponente statt. In größeren Komponenten wurde jedes Feature einzeln getestet. Das Testen erfolgte wie folgt: Die Legacy-Anwendung und die JavaFX-Anwendung werden parallel gestartet, indem beide jeweils über zwei separate Terminalaufrufe ausgeführt werden. Die Aufrufe funktionieren seit der Integration von JavaFX in der Legacy-Anwendung, wie in Abschnitt 4.1.3 beschrieben. Die Anwendung wird über den Aufruf einer `use.bat`-Datei gestartet. Genau nach der `use.bat` folgt entweder ein Parameter `-jfx` für den Start der neuen Version oder kein `-jfx` für den Start der Legacy-Anwendung als Default. Um sofort testen zu können, wird dem Aufruf ein Pfad zu einer `.use`-Datei übergeben, um die Anwendung mit einer Spezifikation zu starten. Die Aufrufe zu beiden Fällen sahen wie folgt aus:

Listing 4.7: Start der Legacy-Anwendung

```
...\bin> .\use.bat ...\examples\...\Fruits.use
```

Listing 4.8: Start der JavaFX-Version

```
...\bin> .\use.bat -jfx ...\examples\...\Fruits.use
```

Nach dem Start beider Anwendungen, wie zum Beispiel in der vorherigen Abbildung 4.3 zu sehen ist, wurde die Funktionalität der neuen Komponente mit der Legacy-Version abgeglichen. In den meisten Fällen traten Fehler und Bugs auf, woraufhin diese behoben wurden und der Prozess des vergleichenden Testverfahrens wiederholt wurde, bis die Funktionalität wie in der Legacy-Komponente gegeben war.

4.6 Integration mit bestehenden Komponenten

Dieser Ansatz zur Integration bestehender Komponenten entstand, um das im Abschnitt 3.1.4 identifizierten Risiko zu vermeiden, dass das Projekt im vorgegebenen Zeitrahmen nicht vollständig migriert werden kann. In diesem Abschnitt wird die Verwendung von `SwingNode` näher beschrieben, welche in Abschnitt 4.3 angesprochen wurde.

Die Idee, bestehende Swing-Komponenten in die JavaFX-Umgebung zu integrieren, sowie die Informationen zu `SwingNode` stammen aus [22]. `SwingNode` ist in JavaFX enthalten und wird verwendet, um Swing-Komponenten in JavaFX-Fenstern einzubetten.

Dieses Vorgehen entspricht dem Grey-Box-Ansatz, welcher in dem Abschnitt der Migrationsstrategien 3.2 vorgestellt wurde. Die Integration von bestehenden Komponenten in die JavaFX-Umgebung bedeutet, dass alles von den Swing-Komponenten verwendet wird, wie vorhanden. Somit können diese zu einem späteren Zeitpunkt in JavaFX migriert werden.

In diesem Projekt kam `SwingNode` bei den komplexeren Komponenten, wie zum Beispiel denen, die UML-Diagramme erstellen, zum Einsatz. Für die Nutzung von `SwingNode` wurde ChatGPT als unterstützendes Tool verwendet. Von ChatGPT stammt folgendes Nutzungsbeispiel für `SwingNode`: Zu Beginn wird ein neues `SwingNode` initialisiert, in welches im folgenden Schritt eine Swing-Komponente eingebettet wird. Nachdem sich die Swing-Komponente im `SwingNode` befindet, wird sie in ein JavaFX-Fenster eingebettet.

Dies ist nur eine Vorlage von ChatGPT, die für die einzelnen Komponenten individuell angepasst wurde.

Die Integration eines `SwingNode` für die Erstellung eines Klassendiagramms in JavaFX sieht wie folgt aus:

Listing 4.9: Methode zur Erstellung eines Klassendiagrams

```
private void createClassDiagram() {

    SwingNode swingNode = new SwingNode();

    // Swing-Komponente Klassendiagramm
    JComponent swingComponent = ...;
    swingNode.setContent(swingComponent);

    createNewWindow("Class diagram", swingNode, DiagramType.CLASS_DIAGRAM
        );

}
```

So ähnlich sehen auch die anderen Stellen im Code aus, in denen `SwingNode` verwendet wurde.

Somit konnte sichergestellt werden, dass im gegebenen Zeitrahmen die Funktionalität aller Komponenten von der Legacy-Anwendung in die JavaFX-Version migriert werden konnte.

4.7 Probleme und Herausforderungen in der Migration

Trotz einer gut entwickelten Migrationsstrategie sowie gründlicher Planung und Vorbereitung konnten Probleme und Herausforderungen in der Umsetzung nicht vermieden werden. In diesem Abschnitt werden die dabei auftretenden Probleme und Herausforderungen vorgestellt.

Eine Herausforderung in der Migration von den GUI-Komponenten lag darin, äquivalent funktionierende Klassen in JavaFX zu finden, da es vorkam, dass es zu einigen Swing-Klassen wie *DesktopPane* keine gleichwertigen, gut funktionierenden Klassen gab. Bei der Suche nach Lösungen wurde ChatGPT befragt und dieses schlug *DesktopPaneFX* vor, welches eine externe Lösung ist und nicht mit JavaFX geliefert wird, oder die eigene

Implementierung eines DesktopPanes in JavaFX. Die Entscheidung fiel auf die eigene Implementierung eines DesktopPanes, aufgrund dessen, dass bei der Verwendung von DesktopPaneFX schnell auffiel, dass diese bei komplexeren Komponenten nicht so performant war wie gewünscht und an ihre Grenzen stieß.

Ein Problem bei einigen gleichwertigen Klassen waren äußerliche GUI-Unterschiede, obwohl die Funktionalität beider gleich war. Dies trat zum Beispiel bei der Verwendung von CheckMenuItem in JavaFX als Äquivalent zu JCheckBoxMenuItem aus Swing auf. Beide tun dasselbe, jedoch werden in der JavaFX-Variante Checkboxen visuell erst dann angezeigt, wenn der Button aktiv ist. Somit besteht ein visueller Unterschied, welcher behoben werden könnte, jedoch wurden diese Arten von Problemen schlichtweg akzeptiert und hingenommen. Um nicht gegen eine Technologie zu arbeiten, sondern mit ihr, wie im Abschnitt 3.1.1 mit „Keep it Simple“ beschrieben wurde.

Die gewählte Testmethode, die im vorherigen Kapitel in Abschnitt 4.5 vorgestellt wurde, erwies sich als suboptimal, weil sie bei Änderungen innerhalb komplexer, verzweigter Komponenten, die alleinstehend getestet wurden, dafür sorgte, dass potenzielle Fehler in abhängigen Komponenten gar nicht oder erst zu einem späteren Zeitpunkt erkannt wurden.

Ein weiteres Problem sind Limitationen, die bei der Nutzung von `SwingNode` erreicht wurden. `SwingNode` ist nicht dafür gemacht, große komplexe Swing-Komponenten einzubetten [22, S. 81]. Es wurde festgestellt, dass beim Versuch, in einem `SwingNode` aus der eingebetteten Swing-Komponente ein neues JOptionPane zu öffnen, Probleme auftreten können. Ein Problem in der Form, dass das neue Fenster entweder nicht geöffnet wurde oder hinter der Anwendung erschien. Dies geschah, weil in der Legacy-Anwendung für die JOptionPanes, welche geöffnet wurden, kein Besitzer übergeben wurde. Dieses Problem konnte behoben werden, indem ein Besitzer gesetzt wurde.

5 Evaluation

Die Evaluation besteht aus der Prüfung, ob die Ziele dieser Arbeit erreicht wurden, gefolgt von den Vorteilen der neuen GUI. Abschließend werden die Herausforderungen und erlangten Erkenntnisse geteilt.

5.1 Erreichte Ziele

Diese Arbeit verfolgte die Ziele, die in Abschnitt 1.2 definiert wurden. Ein Ziel war die Modernisierung einer bestehenden Anwendung, welche in dieser Arbeit am Beispiel der Anwendung USE realisiert wurde, indem die GUI-Technologie von Swing/AWT auf JavaFX migriert wurde. Der Fokus lag jedoch nicht nur auf der Migration selbst, sondern auch an der Herangehensweise und dem Migrationsansatz. Diese Ziele konnten erfolgreich erreicht werden, indem ein fundiertes Wissen über die Anwendung USE aufgebaut wurde, sodass im nächsten Schritt die am besten geeignete Migrationsstrategie gefunden werden konnte. Mithilfe dieser Strategie konnte das Ziel der Migration erreicht werden, wobei alle Funktionalitäten auch nach der Migration enthalten sind.

Während der Migration sind jedoch auch Probleme entstanden, wie in Abschnitt 4.7 dokumentiert. Eines dieser Probleme betrifft die schlechte Auswahl der Teststrategie. Diese gewählte Teststrategie führt dazu, dass einige Funktionalitäten fehlerhaft sein könnten, da diese entweder gar nicht als unvollständig erkannt wurden oder erst zu einem späteren Zeitpunkt erkannt wurden. Dies passierte, weil sie in dem vergleichenden Testverfahren zu diesem Zeitpunkt nicht über die GUI-Oberfläche der Anwendung nachgestellt werden konnten.

In dieser Arbeit wurde ChatGPT als unterstützendes Tool verwendet. Der Einsatzbereich erstreckte sich über das gesamte Projekt hinweg, wie in Abschnitt 4.3 gezeigt wurde. Der KI-gestützte Ansatz diente somit nicht nur zur Unterstützung, sondern spielte eine wichtige Rolle im Ablauf der Migration. Vor der Migration konnten mithilfe von ChatGPT

schnell und einfach vergleichbare Arbeiten gefunden werden, um Ideen für die eigene Migrationsstrategie zu gewinnen. Dabei zeigte sich, dass ChatGPT zwar schnell Literaturvorschläge liefert, diese jedoch häufig ungenau oder nicht nachvollziehbar belegt sind. Dieses Phänomen, das bei der Empfehlung von Literatur auftritt, dass diese ausgedacht oder nicht existent ist, ist auch aus anderen Artikeln bekannt, wie dem Artikel von Alkaissi und McFaiane [6]. Außerdem konnten Werkzeuge und Ansätze identifiziert werden, mit denen die Migration vereinfacht und beschleunigt werden konnte. ChatGPT kann Codeabschnitte gut erklären und ist hilfreich bei der Dokumentation, indem es beispielsweise präzise und brauchbare Kommentare hinzufügt.

Bei komplexeren Aufgaben kann es jedoch auch fehlerhafte Informationen liefern, da ChatGPT oft antwortet, als ob es die richtige Antwort kennen würde, auch wenn es die Fragestellung nicht richtig interpretiert hat. In der Implementierung konnte es trotzdem gut unterstützen. Hier ist anzumerken, dass beide in Abschnitt 4.3 vorgestellten Ansätze für die Migration von Code funktionieren. Der Top-Down-Ansatz kann komplette Lösungen liefern, ist aber eindeutig fehleranfälliger und liefert oft unpräzise Lösungen. Der Bottom-Up-Ansatz ist im Vergleich zeitaufwendiger, dafür aber präziser und weniger fehleranfällig.

Eines der größten Probleme in der Nutzung von ChatGPT liegt darin, dass es sehr oft halluziniert. Insbesondere im Umfeld der Softwareentwicklung, wenn die Codeabschnitte stark mit anderen Klassen gekoppelt sind, die ChatGPT nicht bekannt sind. In diesen Fällen tut ChatGPT so, als würde es diese Klassen kennen, oder denkt sich aus, was die Klassen womöglich machen. Daher können generierte Lösungen unvollständig oder unbrauchbar sein.

Der Begriff *Halluzination* entstand, weil KIs wie ChatGPT Unwahrheiten als Wahrheiten verbreiten können [6].

ChatGPT konnte somit den Prozess der Modernisierung einer Legacy-Anwendung in dieser Arbeit beschleunigen und erleichtern. Jedoch ist eine Voraussetzung für die effektive Nutzung von ChatGPT, dass der Anwender eine Person ist, der die Softwareentwicklung und die zu nutzenden Sprachen nicht fremd sind. Nur dann kann der Benutzer ChatGPT effizient als unterstützendes Tool verwenden und gleichzeitig fehlerhafte und unbrauchbare Lösungen frühzeitig erkennen.

Deshalb wurden alle Aussagen stets kritisch geprüft. Die Informationen, die von ChatGPT kommen, sind daher weiterhin mit Vorsicht zu genießen.

In dieser Arbeit konnte ChatGPT als unterstützendes Tool überzeugen.

Schlussendlich konnten alle Ziele der Arbeit erreicht werden.

5.2 Vorteile der neuen GUI

Die neue GUI bringt für die Anwendung diverse Vorteile mit sich. Die Lösung hat nicht nur auf der GUI-Ebene zu Änderungen geführt, sondern auch in der Softwarearchitektur.

Ein großer Vorteil im Vergleich zur vorherigen GUI-Lösung ist die neue, MVC-ähnliche Struktur in der Softwarearchitektur, die JavaFX mit sich bringt. Diese Struktur entsteht durch die Verwendung von FXML, welche wie in Abschnitt 3.3 beschrieben, die Trennung der Benutzeroberfläche von der Anwendungslogik ermöglicht.

Diese Trennung führt zu einer Separation of Concerns (SoC), einem fundamentalen Aspekt der Softwareentwicklung, also einer klaren Trennung der Aufgaben innerhalb eines Systems. Dieses Vorgehen wird mit dem Ziel angewendet, das System in kleinere, überschaubare Teile aufzuteilen. [10, S. 153 f.]

In der Verwendung von JavaFX bedeutet SoC die Trennung der GUI-relevante Informationen in die FXML-Datei und die Anwendungslogik in die Controller-Klasse. Dadurch wird die Lesbarkeit die Wartbarkeit und Erweiterbarkeit der Anwendung verbessert. Die Wartbarkeit ist gegeben, weil GUI-relevante Änderungen direkt in der FXML-Datei vorgenommen werden können, ohne etwas an der Logik ändern zu müssen. Die Erweiterbarkeit ist gegeben, da Änderungen für GUI-Komponenten nur an einem Ort in der FXML angepasst werden müssen und somit leichter zu verwirklichen sind.

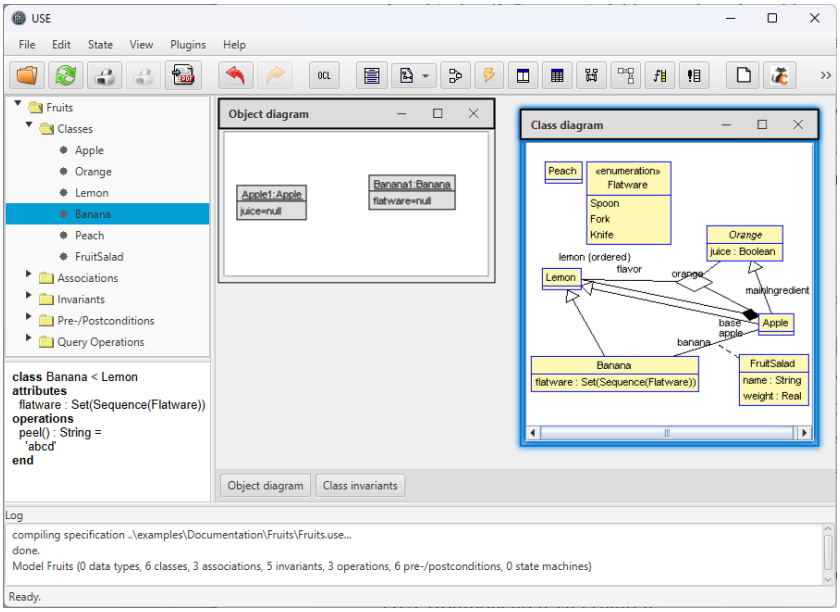


Abbildung 5.1: Die neue GUI-Oberfläche der Anwendung USE

Neben der Änderung in der Softwarearchitektur hat die GUI der Anwendung nach der Migration ein moderneres und für die heutige Zeit ansprechenderes Design, wie in der Abbildung 5.1 im Vergleich zur vorherigen Abbildung 2.1 zu erkennen ist. Die Anwendung verliert außerdem nicht ihre Benutzerfreundlichkeit gegenüber den Verwendern der Legacy-Anwendung, da die Oberflächenstruktur der neuen Anwendung der des alten entspricht, sodass alle Funktionalitäten am selben Ort wiedergefunden werden können. Vor allem bietet die neue GUI eine bessere Lesbarkeit.

Neben der Nutzung von FXML ist es auch möglich, Designänderungen in JavaFX aus einer CSS-Datei zu laden. Um somit viel mehr Möglichkeiten für die Gestaltung von GUI-Komponenten zu erhalten.

5.3 Herausforderungen und Lessons Learned

In Abschnitt 4.7 wurden einige Herausforderungen vorgestellt, die während der Migration auftraten. Einige dieser Herausforderungen führten zu wichtigen Erkenntnissen, die für zukünftige Arbeiten wichtig sind, damit solche Probleme vermieden werden können.

Eine dieser Herausforderungen ist die Findung einer geeigneten Lösung für die Swing-Komponenten in JavaFX. Hier kam es schnell zu falsche Annahmen und Entscheidungen, die viel Zeit kosten können. Wie im Beispiel des `DesktopPane`, bei dem erst mal `DesktopPaneFX` eingebaut wurde, bevor die Entscheidung dann zu einer eigenen Lösung fiel, aufgrund von Performance-Problemen in dieser komplexeren und großen Anwendung. Diese Fehlentscheidungen hätten vermieden werden können, wenn vorher etwas mehr Recherche durchgeführt und die Lösung in einer kleinen Umgebung getestet worden wäre, bevor man es versuchte, in die JavaFX-Anwendung einzubauen. Das `DesktopPane` ist nur eine von mehreren Komponenten, bei denen dieses Problem der Fehlentscheidung aufgetreten ist. Daraus wird die Wichtigkeit der bedachten Entscheidung ersichtlich, da sich diese Fehlentscheidungen aufaddieren und die verlorene Zeit hätte minimieren können, indem man in die Recherche etwas mehr Zeit investiert hätte.

Eine weitere Herausforderung sind die halluzinierten Lösungen von ChatGPT, die in Abschnitt 4.3 angesprochen wurden. Da ChatGPT den Benutzer vom Beginn der Überlegung einer Migration bis zur Finalisierung der Lösung begleitet, kommt es schnell dazu, dass man ihr mit der Zeit mehr vertraut, als man sollte. Es darf jedoch nicht aus den Augen geraten, dass ChatGPT Fehler macht und machen kann, weil sie sehr oft halluziniert. Dies führte oftmals dazu, dass in der migrierten JavaFX-Lösung einige Methoden fehlten, weil ChatGPT diese nicht als notwendig erachtete oder die Zusammenhänge nicht verstand, aufgrund der Tatsache, dass man wegen der Zeilenbeschränkung bei ChatGPT nicht mehrere Klassen übermitteln kann. Daher wurden diese als nicht notwendig betrachteten Methoden nicht migriert, was man bei 1.000 LOC schnell mal übersehen kann. Daher ist es wichtig, die Lösung von ChatGPT immer zu hinterfragen und die Ergebnisse sehr genau zu kontrollieren.

Die oben genannte Herausforderung führt zu einer weiteren Herausforderung, die auftritt, und zwar das Testen dieser JavaFX-Lösungen. Weil ChatGPT halluziniert und somit unvollständige Lösungen liefern kann, konnte es dazu kommen, dass diese fehlenden Stellen der GUI-Komponenten erst beim Testen auffielen. Hier fiel jedoch schnell auf, dass die gewählte Teststrategie in Abschnitt 4.5 suboptimal ist, da die Komponenten bei jeder kleinen Änderung komplett wiederholt getestet werden mussten. Jedoch konnten die gekoppelten Klassen beim Test übersehen werden, sodass es in der GUI-Ebene nur oberflächlich getestet wurde. Augenscheinlich war dann alles gegeben, jedoch fiel im Verlauf der Migration der jeweiligen Komponenten immer wieder auf, dass in vorherigen Komponenten Unvollständigkeiten vorhanden waren und übersehen wurden und im Nachhinein behoben werden mussten, was viel Zeit in Anspruch genommen hat. Daher kommt die

Erkenntnis zustande, dass die Wichtigkeit der Teststrategie unterschätzt wurde und dass diese eine essenzielle Rolle für die Qualität der Lösung spielt. Aus diesem Grund sollte die Teststrategie mit Bedacht gewählt werden.

6 Fazit und Ausblick

Dieser Abschnitt fasst die Arbeit zusammen und bietet einen Ausblick auf potenzielle zukünftige Arbeiten.

6.1 Zusammenfassung der Arbeit

Das Ziel dieser Arbeit war die Modernisierung einer Legacy-Anwendung. Es wurden Anforderungen analysiert, um für die gewählte Anwendung eine passende Migrationsstrategie zu entwickeln. Für die Migration wurde ChatGPT als unterstützendes Tool verwendet.

Damit sichergestellt werden konnte, dass alle Funktionalitäten der Anwendung auch nach der Migration noch vorhanden sind, wurde eine Teststrategie entwickelt.

Das Ziel konnte erfolgreich an der GUI der Anwendung USE umgesetzt werden. Die GUI der Anwendung wurde vollständig von Swing/AWT in JavaFX migriert.

Die gewählte Teststrategie ist für die Modernisierung einer Legacy-Anwendung ungeeignet und kann nicht empfohlen werden.

ChatGPT hatte im Großen und Ganzen einen positiven Einfluss auf das Projekt und kann aus dem Prozess der Softwareentwicklung nicht mehr weggedacht werden. ChatGPT ist ein Werkzeug wie jedes andere und sollte somit auch verwendet werden.

Wie jedes andere Werkzeug gibt es Limitationen und auch Bereiche, in denen es nicht so gut oder nur eingeschränkt gut funktioniert.

Es führte zu einer deutlichen Beschleunigung bei der Migration von einfachen Komponenten der Anwendung, indem es Code generierte, der problemlos übernommen werden konnte.

Bei komplexeren Komponenten konnte es hilfreiche Tipps und Codebeispiele für die Migration mitgeben. Somit haben die Vorteile der Nutzung der ChatGPT die Nachteile überwogen.

Aus dieser Arbeit geht hervor, dass die gewählte Migrationsstrategie der inkrementellen Migration für die Migration von Legacy-Anwendungen sehr gut geeignet ist, da sie die schrittweise Migration der einzelnen Komponenten der Anwendung und das gekapselte Testen dieser ermöglicht. Dies bestätigt auch die Aussage, dass am häufigsten ein inkrementeller Ansatz gewählt wird, um das Risiko in der Migration zu minimieren, welches in Abschnitt 3.2.1 erwähnt wurde und aus dem Forschungsbericht [7] stammt.

6.2 Ausblick

Die GUI der Anwendung USE konnte zwar vollständig migriert werden, jedoch mit dem Kompromiss, dass die GUI der komplexeren Komponenten zunächst mit SwingNodes integriert wurde. Daher könnte eine mögliche Weiterentwicklung so aussehen, dass diese Komponenten vollständig in JavaFX migriert werden.

Außerdem könnte eine bessere Teststrategie erarbeitet werden, die erweiterbar ist und die Anwendung zuverlässig auf die Funktionalitäten testet, ohne dass der Aufwand des Testens, wie in diesem Fall, exponentiell steigt, je mehr Komponenten hinzukommen.

Somit bleibt die Frage offen, wie eine Legacy-Anwendung bei der Migration getestet werden kann, vor allem wenn diese, wie in dieser Anwendung, keine GUI-Tests enthält. Denn wie in der Risikoanalyse in Abschnitt 3.1.4 erwähnt, gibt es in vielen Legacy-Systemen entweder keine oder nur unzureichende Tests.

Aus dieser Arbeit geht die Erkenntnis hervor, dass ein KI-Chatbot wie der in dieser Arbeit verwendete ChatGPT den Prozess der kompletten Softwareentwicklung unterstützen kann, in manchen Bereichen wie der Planung und dem Design etwas besser, in anderen Bereichen wie der Entwicklung hingegen stark abhängig von der Komplexität der Aufgaben. Dennoch kann er in allerlei Softwareentwicklungsprojekten unterstützend eingesetzt werden, mit der Voraussetzung, dass alle Antworten von ChatGPT überprüft werden, um Fehler zu vermeiden, die es machen kann.

In letzter Zeit werden immer mehr KIs in IDEs in Form von Plugins integriert. In diesem Projekt wurde testweise das Plugin von IntelliJ getestet, das leider keine genauso guten

Ergebnisse lieferte wie die Verwendung von ChatGPT für komplexere Fragen. Die nicht zufriedenstellenden Ergebnisse kommen daher, dass dieses Plugin nur Zugriff auf die ausgewählten Klassen oder Codeabschnitte hat, die man der integrierten KI zur Verfügung gestellt hat, für die man Fragen stellt.

Jedoch sind diese Erweiterungen sehr vielversprechend, denn wenn es dazu kommt, dass sie die komplette Codebasis kennen, dann könnte das Problem der Halluzination durch die KI minimiert werden, da die KI dann die Kopplungen der Komponenten kennt und somit Zusammenhänge nicht interpretieren muss. Somit bleibt es für Softwareentwickler spannend, wie die KI sich noch weiter in den Prozess der Softwareentwicklung integrieren wird.

Literaturverzeichnis

[1] *GitHub - openjdk/jfx: JavaFX mainline development — github.com*. https://github.com/openjdk/jfx. – [Accessed 08-07-2025]

[2] *Java AWT vs Java Swing vs Java FX - GeeksforGeeks — geeksforgeeks.org*. https://www.geeksforgeeks.org/java/java-awt-vs-java-swing-vs-java-fx. – [Accessed 09-07-2025]

[3] *JavaFX CSS Reference Guide — openjfx.io*. https://openjfx.io/javadoc/24/javafx.graphics/javafx/scene/doc-files/cssref.html#introscenegraph. – [Accessed 09-07-2025]

[4] *KI in der Softwareentwicklung – Bedeutung und Potenziale*. https://www.explicatis.com/blog/ki-grundlagen-softwareentwicklung. 2024. – Letzte Aktualisierung: 22.04.2024; Letzter Zugriff: 03.07.2025

[5] : *Legacy System: Alte Software modernisieren - Fraunhofer IESE — iese.fraunhofer.de*. https://www.iese.fraunhofer.de/de/leistungen/legacy-systeme.html. 2025

[6] Alkaissi, Hussam ; McFarlane, Samy I.: Artificial Hallucinations in ChatGPT: Implications in Scientific Writing. In: *Cureus* (2023), S. 4. – URL https://www.cureus.com/articles/138667. – Published February 19, 2023

[7] Bragagnolo, Santiago ; Anquetil, Nicolas ; Ducasse, Stéphane ; Seriai, Abderrahmane ; Derras, Mustapha: Software Migration: A Theoretical Framework. / Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille. URL https://inria.hal.science/hal-03171124, 2021. – Research Report

[8] Calà, Ambra ; Lüder, Arndt ; Cachada, Ana ; Pires, Flávia ; Barbosa, José ; Leitão, Paulo ; Gepp, Michael: Migration from traditional towards cyber-physical production systems. In: *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, 2017, S. 1147–1152

[9] Comella-Dorda, Santiago ; Wallnau, Kurt ; Seacord, Robert ; Robert, John: A Survey of Legacy System Modernization Approaches. URL https://insights.sei.cmu.edu/library/a-survey-of-legacy-system-modernization-approaches/, Apr 2000 (CMU/SEI-2000-TN-003). – Forschungsbericht. Accessed: 2025-Jul-7

[10] Daga, A. ; Cesare, Sergio de ; Lycett, M.: Separation of Concerns: Techniques, Issues and Implications. In: *Journal of Intelligent Systems* 15 (2006), 12

[11] Demeyer, Serge ; Ducasse, Stéphane ; Nierstrasz, Oscar: *Object-Oriented reengineering patterns*. – URL https://oorp.github.io/. – Zugriffsdatum: 16-11-2024

[12] Fowler, Martin: *bliki: Strangler Fig*. – URL https://martinfowler.com/bliki/StranglerFigApplication.html. – Zugriffsdatum: 19-03-2025

[13] JavaBeginners.de: *Die Einführung der Modularisierung in Java 9 erlaubt die Untergliederung eines Quelltextes in mehrere strukturelle Einheiten*. 2023. – URL https://javabeginners.de/Grundlagen/Module_verwenden.php. – Letzte Aktualisierung: 24.12.2023; Letzter Zugriff: 09.06.2025

[14] JavaBeginners.de: *Was ist und wozu dient FXML?* 2023. – URL https://javabeginners.de/Frameworks/JavaFX/FXML.php. – Letzte Aktualisierung: 01.04.2023; Letzter Zugriff: 20.03.2025

[15] JetBrains: *IntelliJ IDEA Documentation - Class Diagram*. 2024. – URL https://www.jetbrains.com/help/idea/class-diagram.html. – Zugriffsdatum: 20-03-2025

[16] JetBrains-sceneBuilder: *Configure JavaFX Scene Builder*. 2024. – URL https://www.jetbrains.com/help/idea/opening-fxml-files-in-javafx-scene-builder.html. – Zugriffsdatum: 11-06-2025

[17] Nitin, Vikram: Using AI to Automate the Modernization of Legacy Software Applications. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA : Association for Computing Machinery, 2024 (ASE '24), S. 2514–2517. – URL https://doi.org/10.1145/3691620.3695610. – ISBN 9798400712487

[18] Oracle.com: *JDK 11 Release Notes*. https://www.oracle.com/java/technologies/javase/11-relnote-issues.html

[19] Panitz, Sven E.: *Grafische Benutzeroberflächen mit Swing.* S. 387–414. In: *Java für Teetrinker: Ein funktionaler Zugang zur Programmierung mit Java.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2024. – URL https://doi.org/10.1007/978-3-662-69321-6_11. – ISBN 978-3-662-69321-6

[20] Pereira, Anil L.: 2D Animation of Recursive Backtracking Maze Solution Using JavaFX Versus AWT and Swing. In: *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2020, S. 1787–1793

[21] Project, OpenJFX: *JavaFX and IntelliJ IDEA.* – URL https://openjfx.io/openjfx-docs/#IDE-Intellij. – Zugriffsdatum: 09-06-2025

[22] Robillard, Martin P. ; Kutschera, Kaylee: Lessons Learned While Migrating From Swing to JavaFX. In: *IEEE Software* 37 (2020), Mai, Nr. 3, S. 78–85. – URL http://dx.doi.org/10.1109/MS.2019.2919840. – ISSN 1937-4194

[23] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard.* 1. Heidelberg : dpunkt.verlag GmbH, 2003. – URL https://www.assets.dpunkt.de/openbooks/Spillner_BWSoftwaretest_1A-E-Book.pdf. – ISBN 3-89864-178-3

[24] Team, Altexsoft E.: *Strangler Fig Pattern and Other Methods for Legacy Travel System Migration: Hands-on Experience.* September 2024. – URL https://www.altexsoft.com/blog/strangler-fig-legacy-system-migration/. – Zugriffsdatum: 19-03-2025

[25] Useocl: *USE - UML-based Specification Environment.* – URL https://github.com/useocl/use. – Zugriffsdatum: 20-03-2025

[26] Useocl: *use/manual/main.md at master · useocl/use.* – URL https://github.com/useocl/use/blob/master/manual/main.md. – Zugriffsdatum: 20-03-2025

A Anhang

A.1 Verwendete Hilfsmittel

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

Tabelle A.1: Verwendete Hilfsmittel und Werkzeuge

Tool	Verwendung
ChatGPT	KI-Chatbot verwendet zur sprachlichen Prüfung eigener Texte (Paraphrasierung, Grammatik- und Rechtschreibprüfung).

Erklärung zur selbständigen Bearbeitung

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

Ort	Datum	Unterschrift im Original