University of Applied Sciences
Ravensburg Weingarten

Angewandte Informatik

Projektarbeit

KI-gestützte Erstellung von Softwaretests in der Programmiersprache Java

Verfasser:
Emircan Tutar
Mat.No. 35606

Betreuer:
Prof. Dr. Marius
Hofmeister

Beginn: 01. April 2024
Abgabe: 30. Juni 2024

Inhaltsverzeichnis

1 Einleitung

In der modernen Softwareentwicklung nimmt die Qualitätssicherung durch automatisierte Tests eine sehr große Rolle ein, um effizientere Ergebnisse in Softwareprojekten zur erhalten. Eines der führenden Frameworks im Bereich der Java-Programmierung ist „JUnit“ – dieses Framework ermöglicht die Durchführung von wiederholbaren Tests, damit die Korrektheit einzelner Einheiten des Programmcodes (sog. Units) sichergestellt werden kann. In Verbindung mit ChatGPT4 eröffnen sich neue oder evtl. sogar effizientere Wege, um Testklassen nicht nur zu generieren, sondern auch um die damit verbundenen Herausforderungen und Lösungsansätze zu diskutieren.

In diesem Kontext entwickelt sich die Integration künstlicher Intelligenz in den Prozess der Softwareentwicklung und -testung zu einem vielversprechenden Ansatz, insbesondere das KI-gestützte Modell ChatGPT4. Das Large Language Model ChatGPT4 von Open AI besitzt nämlich die Fähigkeit, komplexe Anforderungen zu verstehen und darauf basierend präzise Testklassen zu generieren, dies stellt einen bedeutsamen Fortschritt in der Programmierwelt dar.

Die Anwendung von ChatGPT4 in der Erstellung von JUnit-Testklassen hat das Potenzial, die Testabdeckung erheblich zu steigern und gleichzeitig den Entwicklungsaufwand für Tests gering zu halten. Durch die Fähigkeit, dynamisch und effektiv auf die ständig wechselnden Anforderungen der Softwareentwicklung zu reagieren, könnte ChatGPT4 eine sehr wichtige Rolle in der Qualitätssicherung einnehmen. Dadurch wäre es möglich, die Lücke zwischen dem Bedarf an hoher Softwarequalität und den begrenzten Ressourcen für das implementieren von manuellen Testmethoden zu überbrücken.

Diese Arbeit zielt darauf ab, das Potenzial von ChatGPT4 zur Unterstützung der automatisierten Testgenerierung in JUnit eingehend zu untersuchen. Dabei wird besonders die erzielbare Testabdeckung (sogenannte Test Coverage), sowie der erforderliche Aufwand für das Prompt-Engineering untersucht. Insgesamt ist das Ziel herauszufinden, inwieweit ChatGPT4 eine revolutionäre Rolle in der Qualitätssicherung einnehmen kann, um den Anforderungen moderner Softwareentwicklungs-Projekte gerecht zu werden.

1.1 Motivation

Die Softwareentwicklung entwickelt sich ständig weiter und erfordert verbesserte Methoden, um die Qualität sicherzustellen. Insbesondere in der Java-Entwicklung ist die Gewährleistung der Softwarequalität durch effiziente Testverfahren essenziell. Automatisierte Tests (wie z.B.Unit-Tests) spielen somit eine entscheidende Rolle bei der Identifizierung und Behebung von Fehlern bereits in frühen Entwicklungsphasen. Trotz des hohen Stellenwerts, den das Testen für die Softwarequalität einnimmt, stellt der damit verbundene Zeit- und Arbeitsaufwand oft eine Herausforderung dar. Vor diesem Hintergrund eröffnet der Einsatz von künstlicher Intelligenz, insbesondere von großen Sprachmodellen (LLMs) wie ChatGPT, neue Möglichkeiten zur Automatisierung und Effizienzsteigerung im Testprozess. Vitor H. Guilherme und Auri M. R. Vincenzi haben in ihrer Arbeit aufgezeigt, dass die von ChatGPT generierten „Test-Sets“ eine mit traditionellen Methoden vergleichbare Qualität aufweisen können, was auf das Potenzial von LLMs zur Verbesserung der Testabdeckung und zur Reduzierung des manuellen Aufwands hinweist [1].

1.2 Zielsetzung

Das Ziel dieser Arbeit besteht darin, die Fähigkeit von ChatGPT zur Generierung von Testklassen zu evaluieren und zu analysieren, inwiefern diese Testklassen zur Verbesserung der Softwarequalität beitragen können. Dabei wird insbesondere der Fokus auf die Code-Coverage und den erforderlichen Aufwand für das Prompt-Engineering gelegt. Es soll untersucht werden, ob und wie ChatGPT den Prozess der Testentwicklung unterstützen und optimieren kann, um effektivere und effizientere Teststrategien für Java-Anwendungen zu ermöglichen.

1.3 Aufbau der Arbeit

Diese Arbeit gliedert sich in sechs Kapitel. Kapitel 1 führt in das Thema ein und erläutert die Motivation sowie die Zielsetzung der Untersuchung. Kapitel 2 legt die theoretischen Grundlagen dar, erklärt die Test-Frameworks wie Junit und TestFX und die Rolle von ChatGPT bei der Testgenerierung. Kapitel 3 beschreibt die methodische Vorgehensweise, die Auswahl der Java-Programme und die Messung der Testabdeckung. In Kapitel 4 und 5 werden die Ergebnisse präsentiert und analysiert. Abschließend fasst Kapitel 6 die wesentlichen Erkenntnisse zusammen.

2 Grundlagen

In diesem Kapitel werden die Grundlagen behandelt, die für das Verstehen der nachfolgenden Arbeit nötig sind. Dieses Kapitel ist in neun Abschnitten unterteilt, um einen Überblick über die wichtigen Konzepte zu geben. Hauptsächlich geht es in den nachfolgenden Punkten um Softwaretests, Test Frameworks und das Large Language Model ChatGPT4.

2.1 Softwaretests

Softwaretests nehmen eine zentrale Rolle in der Qualitätssicherung von Softwareprodukten ein und beanspruchen zwischen 30 % bis 70 % der gesamten Entwicklungszeit. Trotz ihrer essenziellen Bedeutung werden sie häufig unterschätzt oder als Nebentätigkeit angesehen, vor allem, wenn in zeitkritischen Projekten an der für Qualitätssicherung vorgesehenen Zeit Einsparungen vorgenommen werden. Eine solche Herangehensweise führt nicht selten zu Produkten, die beim Kunden „reifen“ und erst nach der Auslieferung ihre Mängel offenbaren, was den Ruf des Produkts und das Vertrauen in die Entwickler ernsthaft gefährden kann [2].

Entgegen der verbreiteten Ansicht, dass das Testen von Software eine monotone und unkreative Tätigkeit sei, erfordert es in Wirklichkeit ein hohes Maß an Sorgfalt, Kreativität und systematischem Denken. Tester stehen vor der Herausforderung, mit minimalem Einsatz von Ressourcen eine maximale Anzahl relevanter Fehler zu identifizieren. Moderne Vorgehensweisen und Werkzeuge haben das Testen zu einer herausfordernden Aufgabe gemacht, die weit über das bloße Abarbeiten vorgegebener Testfälle hinausgeht [2]. Genau hier stellt sich die Frage, ob die Nutzung von Large Language Modellen wie z.B. ChatGPT4, eine effizientere Methode bietet, für die Test-Entwicklung in Java und wenn ja, wie viel Ressourcen dabei eigentlich gespart werden können.

2.2 Teststufen

Teststufen sind verschiedene Ebenen im Testprozess, die darauf abzielen, unterschiedliche Aspekte und Abstraktionsebenen eines Systems zu überprüfen. Jede Teststufe hat spezifische Ziele, Methoden und Werkzeuge, um die Qualität der Software sicherzustellen. Die Haupteststufen umfassen:

- **Unit-Tests:** Überprüfung einzelner Softwarekomponenten, um sicherzustellen, dass sie den Spezifikationen entsprechen und korrekt funktionieren.
- **Integrations-Tests:** Prüfung der Interaktionen zwischen verschiedenen Komponenten oder Modulen, um sicherzustellen, dass sie zusammenarbeiten und die technische Spezifikation erfüllen.

- **UI-Tests:** Tests der Benutzeroberfläche und Benutzerinteraktionen, um die Funktionalität und Benutzerfreundlichkeit der Oberfläche sicherzustellen. Diese Tests stellen sicher, dass die Anwendung korrekt auf Benutzereingaben reagiert.

Jede dieser Teststufen betrachtet das Produkt aus einer anderen Perspektive und auf einer anderen Abstraktionsebene. Entsprechend kommen unterschiedliche Testmethoden, Testwerkzeuge und spezialisiertes Testpersonal zum Einsatz. Diese systematische Unterteilung der Testaktivitäten hilft, die Qualität des Softwareprodukts zu sichern und potenzielle Fehler frühzeitig zu erkennen und zu beheben [4].

2.3 JUnit – ein Framework für Unit-Tests in Java

JUnit ist ein populäres Framework für das Schreiben und Ausführen von Unit Tests in der Java-Programmiersprache. Es bietet eine einfache und zugängliche Methode, um Testfälle zu erstellen, wobei es Annotationen wie `@Test` verwendet, um Methoden zu kennzeichnen, die Testfälle enthalten. JUnit unterstützt auch Testvorbereitung und Aufräumvorgänge mit `@Before` und `@After`, was das Setup für Tests vereinfacht. Die Verwendung von Assertions innerhalb der Testmethoden ermöglicht es Entwicklern, erwartete von tatsächlichen Ergebnissen zu unterscheiden und somit die Zuverlässigkeit der Softwarekomponenten zu überprüfen [5]. JUnit hat sich als ein wesentliches Werkzeug in der Java-Entwicklung etabliert und unterstützt Entwickler nicht nur bei der Fehlererkennung, sondern auch bei der strukturierten Verbesserung der Codequalität durch regelmäßige Tests [2].

2.4 TestFX – ein Framework für UI-Tests in JavaFX

UI-Tests (User Interface Tests) mit TestFX sind automatisierte Tests, die die Benutzeroberfläche einer JavaFX-Anwendung überprüfen. TestFX ermöglicht es, Benutzeraktionen wie Klicks, Tastatureingaben und andere Interaktionen zu simulieren, um die Funktionalität und das Verhalten der Oberfläche zu testen.

2.5 Code-Coverage

Code-Coverage ist ein Maß für den Umfang des Codes, der durch Tests abgedeckt wird. Es zeigt, welche Teile des Codes während der Testausführung tatsächlich ausgeführt wurden, und hilft dabei, Bereiche zu identifizieren, die möglicherweise ungetestet bleiben.

2.5.1 Methoden Coverage

Methoden-Coverage ist eine spezifische Art der Code-Coverage, die misst, wie viele Methoden innerhalb eines Programms durch Tests abgedeckt sind.

Es gibt an, ob alle Methoden mindestens einmal aufgerufen wurden.

2.5.2 Branch-Coverage

Branch-Coverage ist eine weitere Art der Code-Coverage, die überprüft, ob alle möglichen Wege (Branches) durch bedingte Anweisungen in einem Programm getestet wurden. Sie stellt sicher, dass sowohl die wahren als auch die falschen Pfade von Bedingungen abgedeckt sind.

2.6 Getestete Funktionalität

Im Kontext dieses Projekts bezieht sich der Begriff 'getestete Funktionalität' auf die spezifischen Aspekte und Komponenten des Java-Codes, die durch die generierten Testklassen überprüft werden. Dies umfasst eine Vielzahl von Elementen, Methoden, Konstrukten und Kontrollstrukturen innerhalb der Klassen, die getestet werden, um sicherzustellen, dass sie korrekt funktionieren und erwartungsgemäß interagieren. Dabei werden auch die Interaktionen zwischen verschiedenen Komponenten getestet, um sicherzustellen, dass sie reibungslos zusammenarbeiten. Ein weiterer wichtiger Aspekt ist die Validierung von Eingabedaten sowie die Fehler- und Ausnahmebehandlung.

2.7 Testgenerierung mit ChatGPT4

ChatGPT4, das Large Language Model von OpenAI, kann zur Automatisierung der Generierung von Testfällen eingesetzt werden. Durch seine Fähigkeit, natürliche Sprache zu verstehen, kann ChatGPT4 automatisch Testfälle aus Softwarebeschreibungen und Spezifikationen extrahieren. Dies kann den gesamten Testprozess erheblich beschleunigen und ermöglicht es Teams, sich auf andere Aufgaben zu konzentrieren. Die Nutzung von ChatGPT4 in diesem Bereich kann die Effizienz der Testentwicklung steigern, indem schnell umfangreiche und präzise Testdatensätze erstellt werden, die manuell so schnell schwer zu erreichen wären.

2.8 Promptengineering

Promptengineering bezieht sich auf die Wissenschaft, Anfragen oder Eingabeaufforderungen an ein KI-Modell wie ChatGPT so zu formulieren, dass sie die gewünschten und optimalen Antworten liefern. Durch das sorgfältige Gestalten der Prompts können spezifische Informationen abgerufen, die Genauigkeit der Antworten verbessert und unerwünschte Ausgaben minimiert werden. Es ist ein iterativer Prozess, der ständige Anpassungen und Tests erfordert, um die besten Ergebnisse zu erzielen [6].

2.9 Anfrageprompt

Ein Anfrageprompt ist eine spezifische Art von Eingabe, die an ein KI-Modell gesendet wird, um eine bestimmte Aufgabe zu erfüllen oder Informationen zu erhalten. Im Kontext der Testgenerierung beschreibt ein Anfrageprompt detailliert die Anforderungen an die zu erstellenden Tests, einschließlich der zu testenden Funktionen und des gewünschten Testumfangs. Die Qualität des Anfrageprompts spielt eine entscheidende Rolle bei der Generierung relevanter und nützlicher Testergebnisse.

3 Erstellen von Testklassen mit ChatGPT4

In diesem Kapitel wird die Vorgehensweise zur Generierung und Analyse von Tests mithilfe von ChatGPT detailliert beschrieben. Es umfasst die Erläuterung des Projektaufbaus, die Testgenerierung durch Anfrageprompts, die Ausführung und Analyse der Tests in IntelliJ sowie das Promptengineering zur Verbesserung der Coverage und Lösung von Fehlern.

3.1 Sprache und Tools

Das Projekt verwendet die Programmiersprache Java und das Entwicklungs-Framework JUnit für Unit-Tests. Für JavaFX Anwendungen werden GUI-Tests mit TestFX durchgeführt [3]. Die Entwicklungsumgebung ist IntelliJ IDEA, die eine umfassende Unterstützung für diese Tools bietet. Zusätzlich wird Maven als Build- und Projektmanagement-Tool verwendet, um die Abhängigkeiten und das Build-System zu verwalten.

3.2 Vorgehensweise für die Testgenerierung

3.2.1 Programmcode

Der bestehende Programmcode wird in gut aufgeteilten Klassen und Methoden organisiert, die den Kern der Anwendung darstellen. Dieser Code wird Klassen- oder Methodenweise an ChatGPT im Anfrageprompt präsentiert, um die Grundlage für die Testgenerierung zu schaffen.

3.2.2 ChatGPT Projektaufbau erklären

Zunächst wird ChatGPT der Aufbau des Projekts erklärt. Dies umfasst die verwendete Programmiersprache, die eingesetzten Tools und den vorhandenen Programmcode. Hierbei ist es wichtig, eine klare und präzise Beschreibung zu liefern, um sicherzustellen, dass ChatGPT die Struktur und die Anforderungen des Projekts vollständig versteht.

3.2.3 Testgenerierung durch Anfrageprompt

Auf Basis des erklärten Projektaufbaus wird ein spezifischer Anfrageprompt an ChatGPT formuliert, um Testfälle zu generieren. Der Prompt beschreibt detailliert die zu testenden Funktionen und die Art der benötigten Tests. ChatGPT erstellt daraufhin die entsprechenden Testklassen und -methoden. In ChatGPT4 besteht die Möglichkeit, eine komplette Datei als Anhang mit dem Prompt zu übergeben. Dies ermöglicht es, eine .java-Datei, die die zu testende Klasse enthält, direkt in den Anfrageprompt einzubinden. Der Anfrageprompt fordert ChatGPT auf, eine Testklasse zu generieren, die eine sehr hohe Code-Coverage erzielt. Auf diese Weise wird sichergestellt, dass die

generierten Tests effektiv sind, als auch der Inhalt passend zur Java-Klasse ist.

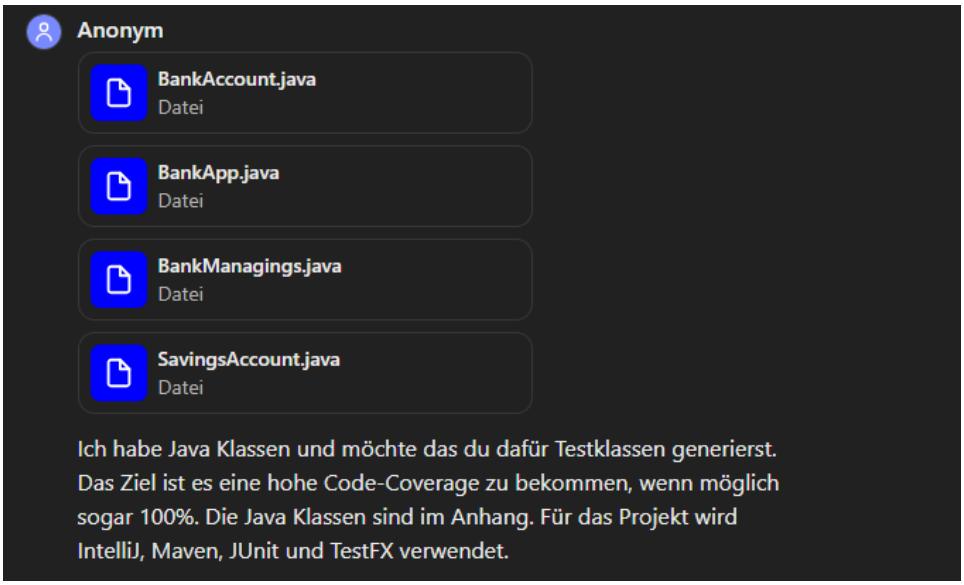

Abbildung 1: Beispiel für einen Anfrageprompt in ChatGPT

Wie in Abbildung 1 gezeigt, enthält der Anfrageprompt eine .java-Datei, in der die zu testende Klasse definiert ist, und fordert ChatGPT auf, eine Testklasse mit hoher Code-Coverage zu generieren.

3.2.4 Analyse der Code Coverage

Die generierten Testklassen werden in IntelliJ IDEA ausgeführt. Dabei wird die Code Coverage analysiert, um zu bestimmen, welche Teile des Codes durch die Tests abgedeckt sind. Die Analyse umfasst sowohl die Methoden Coverage als auch die Branch Coverage:

- **Methoden Coverage:** Diese Analyse misst, wie viele der definierten Methoden im Code durch die Tests abgedeckt sind. Sie gibt Aufschluss darüber, ob alle Methoden mindestens einmal getestet wurden.
- **Branch Coverage:** Diese Analyse überprüft, ob alle möglichen Verzweigungen (Branches) in den bedingten Anweisungen des Codes getestet wurden. Es wird festgestellt, ob sowohl die wahren als auch die falschen Pfade von Bedingungen abgedeckt sind.

Zusätzlich werden auftretende Fehler und fehlgeschlagene Tests dokumentiert, um eine Bewertung der Testqualität und der Abdeckung zu ermöglichen. Diese Analyse hilft dabei, Bereiche des Codes zu identifizieren, die noch ungetestet sind oder in denen Fehler auftreten.

3.2.5 Promptengineering zur Verbesserung der Coverage und Fehlerbehebung

Basierend auf der Analyse der ersten Testläufe werden neue Anfragen an ChatGPT formuliert, um die Testabdeckung zu verbessern und aufgetretene Fehler zu beheben. Dieses iterative Vorgehen, bekannt als Promptengineering, zielt darauf ab, die Qualität und Abdeckung der Tests schrittweise zu optimieren.

Die Methode, die dabei verwendet wird, umfasst die Übermittlung der getesteten Java-Klasse und der dazugehörigen Testklasse als Anhang im Prompt. Zusätzlich werden die in der IDE aufgetretenen Fehlermeldungen eingefügt, zusammen mit einer kurzen Erklärung der Ursache dieser Fehler. Darüber hinaus werden die nicht getesteten Methoden und Branches erwähnt, mit einer Aufforderung, diese in die Tests zu integrieren.

Falls nötig, wird dieser Prozess wiederholt, bis die Tests fehlerfrei sind und die höchstmögliche Code Coverage erreicht ist. Dieses detaillierte und strukturierte Vorgehen stellt sicher, dass ChatGPT alle relevanten Informationen erhält, um effektive Testfälle zu generieren, die die Code Coverage maximieren und bestehende Fehler beheben.

3.2.6 Ausführen der verbesserten Testklassen und Analyse der Ergebnisse

Die verbesserten Testklassen werden erneut in IntelliJ IDEA ausgeführt. Eine erneute Analyse der Code Coverage wird durchgeführt, um die Effektivität der Änderungen zu bewerten. Ziel ist es, eine möglichst hohe Testabdeckung und Fehlerfreiheit zu erreichen.

3.3 Teststufen

Im Rahmen dieses Projekts werden verschiedene Teststufen angewendet, die zur erreichen einer hohen Code-Coverage nötig sind. Diese werden in den Folgenden Kapiteln näher erläutert.

3.3.1 Unit-Tests

Unit-Tests werden isoliert für einzelne Methoden und Klassen durchgeführt, hauptsächlich unter Verwendung von JUnit. In diesem Projekt werden die Unit-Tests erstellt, um die Funktionalität einzelner Methoden und Klassen zu überprüfen. JUnit wird dabei eingesetzt, um sicherzustellen, dass jede Komponente korrekt arbeitet und die Anforderungen erfüllt.

3.3.2 UI-Tests

Für die Benutzeroberfläche und Benutzerinteraktionen werden spezifische Tests mit TestFX durchgeführt. In diesem Projekt liegt der Fokus besonders auf den Controllern, um sicherzustellen, dass die GUI wie erwartet funktioniert und alle Benutzeraktionen korrekt verarbeitet werden. TestFX ermöglicht es, die Interaktion mit der Benutzeroberfläche zu automatisieren und zu überprüfen.

3.3.3 Integrationstests

Integrationstests überprüfen die Interaktionen zwischen verschiedenen Klassen und Modulen des Systems. In diesem Projekt dienen Integrationstests dazu, sicherzustellen, dass die verschiedenen Komponenten korrekt zusammenarbeiten und die definierten Schnittstellen einwandfrei funktionieren. Ziel dieser Teste ist es, Integrationsprobleme frühzeitig zu erkennen und zu beheben.

3.3.4 Mocking

Mocking wird in diesem Projekt nur einfach angewendet, da das Generieren und Untersuchen von Mocking-Szenarien komplex und zeitaufwändig ist. Der Fokus liegt daher weniger auf umfangreichem Mocking und der Analyse davon.

4 Evaluation der Testgenerierung

In diesem Kapitel werden die Ergebnisse der durchgeführten Tests detailliert dargestellt und analysiert. Es umfasst die drei Java-Projekte, die zur Testgenerierung mit ChatGPT verwendet werden.

4.1 Projekt 1: Einfaches Beispiel

4.1.1 Projektbeschreibung

In diesem Abschnitt wird ein praktisches Beispiel zur Erstellung von Testklassen mit ChatGPT4 anhand eines Inventory-Manager-Systems vorgestellt. Das System besteht aus zwei Hauptkomponenten: der ‘Product‘-Klasse und der ‘InventoryManagement‘-Klasse.

Die Auswahl der ‘Product‘- und ‘InventoryManagement‘-Klassen basiert auf ihrer Relevanz und ihrer typischen Anwendung in vielen realen Projekten. Die ‘Product‘-Klasse repräsentiert ein grundlegendes Datenmodell, das häufig in verschiedenen Anwendungen verwendet wird, um Produkte oder Artikel zu beschreiben. Sie enthält typische Attribute wie Produkt-ID, Name, Preis, Kategorie und Lagerbestand, die in vielen Geschäftsanwendungen relevant sind.

Die ‘InventoryManagement‘-Klasse wird ausgewählt, weil sie grundlegende Funktionen zur Verwaltung von Produktdaten in einer Liste bietet. Dies umfasst das Hinzufügen, Entfernen und Suchen von Produkten sowie die Durchführung von Operationen wie Sortieren und Filtern. Diese Klasse stellt ein gutes Beispiel für eine Geschäftslogik dar, die in vielen Anwendungen implementiert wird, und bietet daher eine geeignete Grundlage für die Demonstration der Testgenerierung mit ChatGPT4. Der Fokus dieses Projekts liegt auf einfachen Unit-Tests.

4.1.2 Die Product-Klasse

Die ‘Product‘-Klasse repräsentiert ein einzelnes Produkt mit Eigenschaften wie Produkt-ID, Name, Preis, Kategorie und Lagerbestand. Jedes Produkt wird durch eine eindeutige ID identifiziert. Die Klasse bietet einen Konstruktor, Update-Methoden, sowie Getter- und Setter-Methoden.

Listing 1: Product Klasse

```
public class Product {
      private int productId;
      private String name;
      private double price;
      private String category;
      private int stockQuantity;

      public Product(int productId, String name, double price,
          String category, int stockQuantity) {
            this.productId = productId;
            this.name = name;
            this.price = price;
            this.category = category;
            this.stockQuantity = stockQuantity;
      }

      // Getter-Methoden
      public int getProductId() { return productId; }
      public String getName() { return name; }
      public double getPrice() { return price; }
      public String getCategory() { return category; }
      public int getStockQuantity() { return stockQuantity; }

      // Setter-Methoden
      public void setProductId(int productId) { this.productId =
          productId; }
      public void setName(String name) { this.name = name; }
      public void setPrice(double price) { this.price = price; }
      public void setCategory(String category) { this.category =
          category; }
      public void setStockQuantity(int stockQuantity) { this.
          stockQuantity = stockQuantity; }

      // Update-Methoden
      public void updatePrice(double newPrice) { this.price =
          newPrice; }
      public void updateStockQuantity(int newQuantity) { this.
          stockQuantity = newQuantity; }
}
```

4.1.3 Die InventoryManagement-Klasse

Die ‘InventoryManagement‘-Klasse verwaltet eine Sammlung von ‘Product‘-Objekten. Sie bietet Methoden zum Hinzufügen und Entfernen von Produkten sowie Funktionen, um Produkte nach verschiedenen Kriterien wie Name und Kategorie zu suchen. Zusätzliche Funktionen ermöglichen es, die Produktliste nach Preis zu sortieren oder Produkte unter einem bestimmten Preis zu filtern.

Listing 2: InventoryManagement Klasse

```
public class InventoryManagement {
      public List<Product> products;

      public InventoryManagement() {
            this.products = new ArrayList<>();
      }

      public void addProduct(Product product) {
            products.add(product);
      }

      public boolean removeProduct(int productId) {
            for (int i = 0; i < products.size(); i++) {
                  if (products.get(i).getProductId() == productId)
                        {
                         products.remove(i);
                         return true;
                  }
            }
            return false;
      }

      public Product findProductByName(String name) {
            for (Product product : products) {
                  if (product.getName().equalsIgnoreCase(name)) {
                        return product;
                  }
            }
            return null;
      }

      public List<Product> getProductsInCategory(String category)
          {
            List<Product> filteredProducts = new ArrayList<>();
            for (Product product : products) {
                  if (product.getCategory().equalsIgnoreCase(
                      category)) {
                        filteredProducts.add(product);
                  }
            }
            return filteredProducts;
```

```
    }

    public void sortByPriceAscending() {
        products.sort((p1, p2) -> Double.compare(p1.getPrice()
            , p2.getPrice()));
    }

    public List<Product> getProductsBelowPrice(double priceLimit
        ) {
        List<Product> filteredProducts = new ArrayList<>();
        for (Product product : products) {
            if (product.getPrice() < priceLimit) {
                filteredProducts.add(product);
            }
        }
        return filteredProducts;
    }

    public double calculateTotalInventoryValue() {
        double totalValue = 0;
        for (Product product : products) {
            totalValue += product.getPrice() * product.
                getStockQuantity();
        }
        return totalValue;
    }

    public void applyCategoryDiscount(String category, double
        discountPercentage) {
        for (Product product : products) {
            if (product.getCategory().equalsIgnoreCase(
                category)) {
                double newPrice = product.getPrice() - (
                    product.getPrice() *
                    discountPercentage / 100);
                product.updatePrice(newPrice);
            }
        }
    }

    public void updateProduct(int productId, double newPrice,
        int newQuantity) {
        for (Product product : products) {
            if (product.getProductId() == productId) {
                product.updatePrice(newPrice);
                product.updateStockQuantity(newQuantity);
            }
        }
    }
}
```

4.1.4 Generierung von Testklassen mit ChatGPT4

ChatGPT4 wurde verwendet, um Testklassen für die ‘Product‘-Klasse und die ‘InventoryManagement‘-Klasse zu generieren (siehe Abb. 2). Der Fokus lag dabei auf der Überprüfung der Funktionalität der Methoden unter verschiedenen Bedingungen.

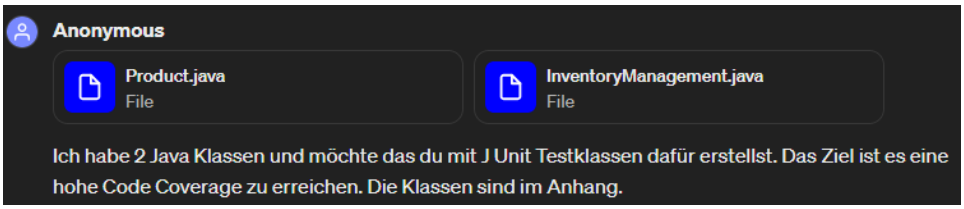

Abbildung 2: Beispiel für den Anfrageprompt in ChatGPT

Die Konversation mit ChatGPT für das generieren der Initialen Test-Klassen ist im **Anhang A.4 verfügbar.**

1. Generierte Testklasse für die Product-Klasse
Die folgende Testklasse für die ‘Product‘-Klasse wird von ChatGPT4 generiert. Diese Testklasse überprüft die grundlegenden Methoden der ‘Product‘-Klasse, einschließlich Getter- und Setter-Methoden sowie Methoden zur Aktualisierung von Preis und Lagerbestand:

Listing 3: Testklasse für Product

```
public class ProductTest {
	private Product product;

	@Before
	public void setUp() {
		product = new Product(1, "Apple", 0.50, "Fruit", 100);
	}

	@Test
	public void testGetters() {
		assertEquals(1, product.getProductId());
		assertEquals("Apple", product.getName());
		assertEquals(0.50, product.getPrice(), 0.001);
		assertEquals("Fruit", product.getCategory());
		assertEquals(100, product.getStockQuantity());
	}

	@Test
	public void testSetters() {
		product.setProductId(2);
		assertEquals(2, product.getProductId());
```

```
            product.setName("Banana");
            assertEquals("Banana", product.getName());

            product.setPrice(0.70);
            assertEquals(0.70, product.getPrice(), 0.001);

            product.setCategory("Snack");
            assertEquals("Snack", product.getCategory());

            product.setStockQuantity(150);
            assertEquals(150, product.getStockQuantity());
      }

      @Test
      public void testUpdatePrice() {
            product.updatePrice(0.60);
            assertEquals(0.60, product.getPrice(), 0.001);
      }

      @Test
      public void testUpdateStockQuantity() {
            product.updateStockQuantity(120);
            assertEquals(120, product.getStockQuantity());
      }
}
```

2. Generierte Testklasse für die InventoryManagement-Klasse
Die folgende Testklasse für die 'InventoryManagement'-Klasse wird von ChatGPT4 erstellt. Diese Testklasse testet die Methoden zur Verwaltung von Produkten, einschließlich Hinzufügen, Entfernen, Suchen und Sortieren von Produkten, sowie die Berechnung des Gesamtwerts des Inventars:

Listing 4: Testklasse für InventoryManagement

```
public class InventoryManagementTest {
      private InventoryManagement inventory;
      private Product apple;
      private Product banana;

      @Before
      public void setUp() {
            inventory = new InventoryManagement();
            apple = new Product(1, "Apple", 0.50, "Fruit", 100);
            banana = new Product(2, "Banana", 0.30, "Fruit", 150);
            inventory.addProduct(apple);
            inventory.addProduct(banana);
      }

      @Test
      public void testAddProduct() {
```

```
        Product orange = new Product(3, "Orange", 0.60, "Fruit
            ", 80);
        inventory.addProduct(orange);
        assertEquals(3, inventory.products.size());
    }

    @Test
    public void testRemoveProduct() {
        assertTrue(inventory.removeProduct(1));
        assertFalse(inventory.removeProduct(4));
        assertEquals(1, inventory.products.size());
    }

    @Test
    public void testFindProductByName() {
        assertEquals(apple, inventory.findProductByName("Apple
            "));
        assertNull(inventory.findProductByName("Grape"));
    }

    @Test
    public void testGetProductsInCategory() {
        List<Product> fruits = inventory.getProductsInCategory
            ("Fruit");
        assertEquals(2, fruits.size());
        // Test with no products in the category
        List<Product> snacks = inventory.getProductsInCategory
            ("Snack");
        assertTrue(snacks.isEmpty());
    }

    @Test
    public void testSortByPriceAscending() {
        inventory.sortByPriceAscending();
        assertEquals(banana, inventory.products.get(0));
    }

    @Test
    public void testGetProductsBelowPrice() {
        List<Product> cheapFruits = inventory.
            getProductsBelowPrice(0.40);
        assertEquals(1, cheapFruits.size());
        assertEquals(banana, cheapFruits.get(0));
    }

    @Test
    public void testCalculateTotalInventoryValue() {
        double value = inventory.calculateTotalInventoryValue
            ();
        assertEquals(95.0, value, 0.001);
    }

    @Test
    public void testApplyCategoryDiscount() {
```

```
            inventory.applyCategoryDiscount("Fruit", 10);
            assertEquals(0.45, apple.getPrice(), 0.001);
            assertEquals(0.27, banana.getPrice(), 0.001);
            // Test with no applicable discount due to wrong
                category
            inventory.applyCategoryDiscount("Vegetable", 10);
            assertEquals(0.45, apple.getPrice(), 0.001);
            assertEquals(0.27, banana.getPrice(), 0.001);
      }

      @Test
      public void testUpdateProduct() {
            inventory.updateProduct(1, 0.55, 110);
            assertEquals(0.55, apple.getPrice(), 0.001);
            assertEquals(110, apple.getStockQuantity());
      }
}
```

4.1.5 Analyse und Ergebnisse

A) Fehlerrate und Fehlerbehebung

Bei der ersten Implementierung der Methode ‘testCalculateTotalInventoryValue‘ durch ChatGPT4 wird ein Fehler entdeckt, der durch die Assert-Ausgabe aufgedeckt wird. Der Fehler entsteht aufgrund einer falschen Berechnung des erwarteten Werts (siehe Abb. 3). Nachdem dieser Fehler an ChatGPT4 gemeldet wurde, identifiziert das System die Ursache und korrigiert den Fehler schnell (siehe Abb. 4). Die Korrektur ist unkompliziert und die Methode funktioniert nach der Anpassung fehlerfrei. Dies zeigt das ChatGPT4, auf Feedback korrekt reagiert und notwendige Anpassungen vornehmen kann wenn es um einfache Unit-Tests geht.

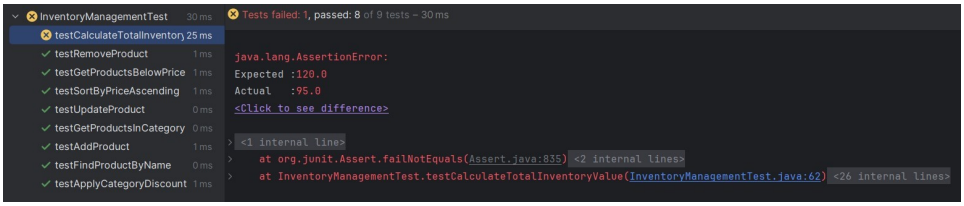

Abbildung 3: Fehlermeldung in der Assert-Ausgabe

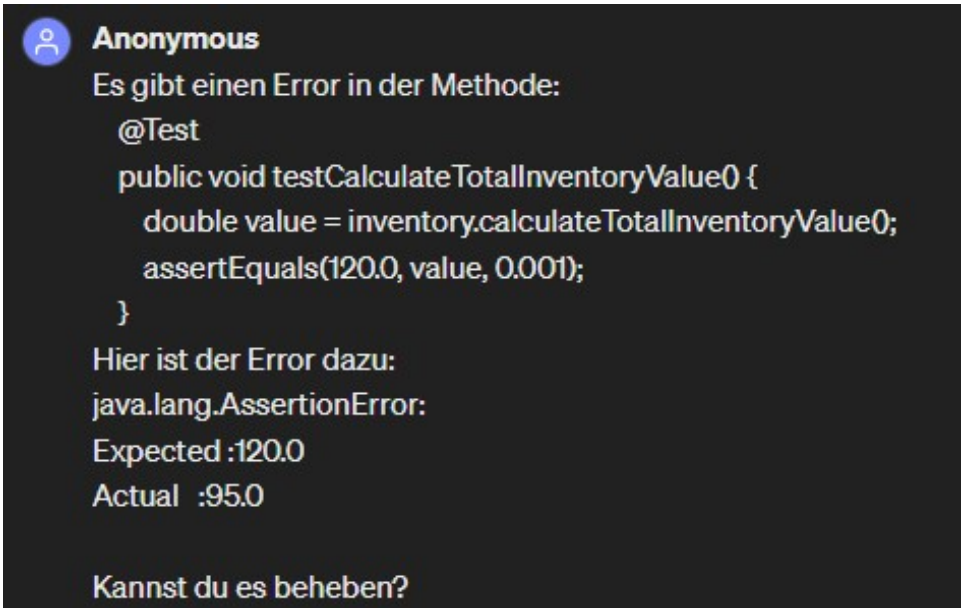

Abbildung 4: Korrekturanfrage an ChatGPT4

B) CodeCoverage
Die initiale Testgenerierung durch ChatGPT4 liefert beeindruckende Ergebnisse hinsichtlich der Code-Coverage:

- **Methoden Coverage**: 100%
- **Branch-Coverage**: 92%

Nach der Aufforderung an ChatGPT4, die Branch-Coverage zu verbessern, werden die fehlenden Testfälle ergänzt. Dadurch wird schließlich auch bei der Verzweigungsabdeckung eine Abdeckung von 100% erreicht. Dies zeigt, dass ChatGPT4 in der Lage ist, eine vollständige Testabdeckung in einfachen Java-Klassen zu erzielen (siehe Abb.5).

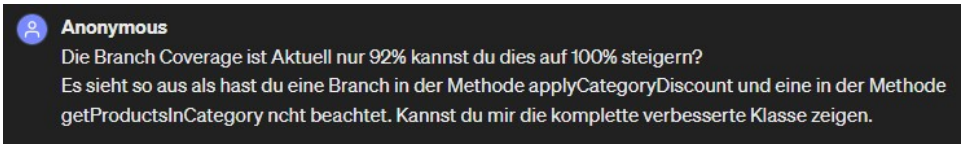

Abbildung 5: Prompt zur Verbesserung der Branch-Coverage

C) Aufwand für das Promptengineering
Der Aufwand für das Prompt-Engineering beim Erstellen der Testklassen ist sehr gering. Die Java-Klassen werden ChatGPT4 bereitgestellt, und das Modell leitet relevante Testfälle daraus ab und implementiert sie ohne große Probleme. Für die initiale Implementierung der Testklassen sind keine komplexen oder detaillierten Anweisungen somit nötig. Wenn Anpassungen benötigt werden, hilft bereits eine einfache Aufforderung, Fehler zu korrigieren und die Testabdeckung zu erweitern (siehe Abb. 4 und 5). ChatGPT4 zeigt sich dabei anpassungsfähig und nimmt notwendige Änderungen an den Testklassen ohne großen Aufwand vor.

Es sollte jedoch erwähnt werden, dass die Ergebnisqualität von ChatGPT direkt mit der Klarheit und Genauigkeit der gestellten Prompts zusammenhängt. Es ist deutlich, dass ChatGPT erst dann effektive Ergebnisse liefert, wenn es klar versteht, was von ihm erwartet wird. Das bedeutet, dass ein grundlegendes Verständnis von Java, JUnit und Testimplementierung für den Prompt-Engineer unverzichtbar ist. Dieses Wissen ist notwendig, um dem Modell die erforderlichen Informationen klar und verständlich zu übermitteln, damit es die Aufgabe korrekt ausführen kann.

D) Getestete-Funktionalität
Die von ChatGPT4 generierten Testklassen decken eine Vielzahl von Funktionalitäten ab und zeigen eine hohe Zuverlässigkeit in der Testausführung. Die Tests umfassen folgende Funktionalitäten:

Getestete Funktionalität	Positive Erkenntnisse	Negative Erkenntnisse
Getter- und Setter	Werden vollständig durchgetestet	Keine
Konstruktoren	Werden vollständig durchgetestet	Keine
IF-ELSE Statements	Gute Abdeckung, außer bei NULL-Fällen	Zweige werden öfters übersprungen, sowie z.B. NULL-Fälle
Updatemethoden	Werden vollständig durchgetestet	Keine
Array Listen	Werden vollständig durchgetestet	Keine
Such-, Sortier- und Filter Algorithmen	Fast Problemlose Überprüfung der Funktionen	Der NULL-Fall wird nicht immer getestet
Berechnungsfunktionen	Probleme werden schnell nachgerechnet und korrigiert	Manchmal einfache Rechenfehler, muss nachgeprüft werden von einer Person
Iterations-Prozesse	Problemlose Überprüfung von Schleifen	Keine
Einfache Fehlerbehandlungen	Werden vollständig durchgetestet	Keine

Abbildung 6: Tabelle der Getestete Funktionalität

Die von ChatGPT4 erstellten Testklassen zeigen, dass sie die Hauptfunktionen der Java-Klassen gut abdecken. Die Tests für Getter und Setter, Konstruktoren sowie Update-methoden laufen durchweg ohne Probleme. Komplexere Funktionalitäten wie die Bearbeitung von Listen und die Ausführung von Such-, Sortier- und Filteroperationen werden ebenfalls erfolgreich geprüft. Eine Schwachstelle zeigt sich jedoch bei der Handhabung von Null-Fällen innerhalb von IF-ELSE-Statements und bei einfachen Berechnungsfunktionen. Diese können jedoch durch die Nachbesserungsfähigkeit von ChatGPT4 schnell behoben werden, indem man ihn darauf hinweist.

4.1.6 Fazit

Zusammenfassend zeigt sich für einfache Testklassen und Unit-Tests, dass ChatGPT4 eine effiziente Unterstützung bei der Generierung von Testklassen für Java-Projekte bietet. Die hohe Testabdeckung und die Fähigkeit zur schnellen Fehlerbehebung und Anpassung machen ChatGPT4 zu einem starken Werkzeug im Testentwicklungsprozess. Trotz kleinerer Schwächen, kann ChatGPT4 durch gezieltes Prompt-Engineering hervorragende Ergebnisse erzielen und diese dienen eigentlich immer als gute Vorlage für einfache Unit-Tests.

4.2 Projekt 2: Komplexeres Beispiel

4.2.1 Projektbeschreibung

In diesem Abschnitt wird ein komplexeres Beispiel zur Erstellung von Testklassen mit ChatGPT4 vorgestellt. Das System besteht aus mehreren Klassen, darunter ‘BankAccount‘, ‘BankApp‘, ‘BankManagings‘ und ‘SavingsAccount‘. Diese Klassen repräsentieren verschiedene Aspekte eines Bankmanagement-Systems.

Die Auswahl dieser Klassen basiert auf ihrer Fähigkeit, verschiedene fortgeschrittene Programmierkonzepte und Funktionalitäten zu demonstrieren, einschließlich Rekursion, verschachtelten Schleifen, Exception-Handling, Switch-Case-Strukturen, Lambda-Ausdrücken, Enums, Vererbung, Zustandautomaten und JavaFX-Komponenten. Diese Vielfalt an Programmierkonzepten macht sie zu einem idealen Kandidaten, um die Leistungsfähigkeit von Chat-GPT4 bei der Generierung von Testklassen zu demonstrieren. Der Fokus dieses Projekts liegt auf komplexeren Unit-Tests und einfachen UI-Tests (Proof of Work).

Das UML-Diagramm (siehe Abb. 7) zeigt die Struktur des Bankmanagement-Systems, während das Sequenzdiagramm (siehe Abb. 8) die Interaktionen zwischen den Klassen verdeutlicht.

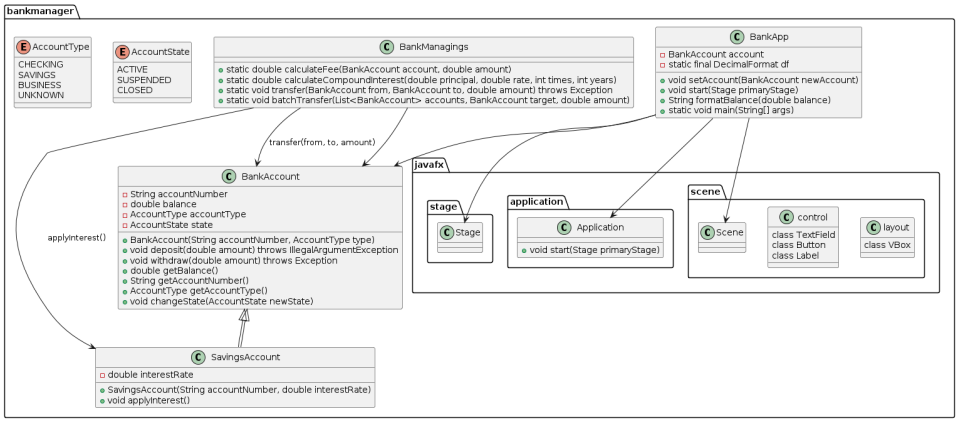

Abbildung 7: UML-Diagramm des Bankmanagement-Systems

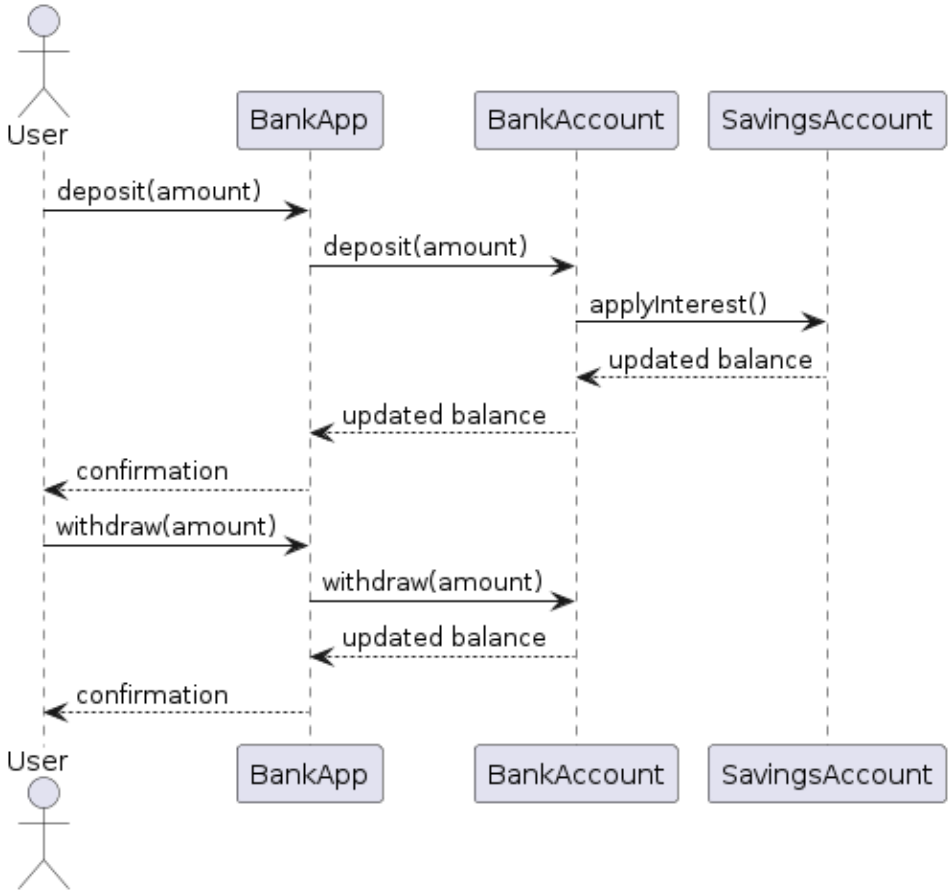

Abbildung 8: Sequenzdiagramm des Bankmanagement-Systems

4.2.2 Die BankAccount-Klasse

Die ‘BankAccount‘-Klasse repräsentiert ein Bankkonto mit Attributen wie Kontonummer, Kontostand, Kontotyp und Kontostatus. Die Klasse bietet Methoden zum Einzahlen und Abheben von Geld, sowie zum Ändern des Kontostatus. Die wichtigsten Methoden sind:

- **deposit(double amount)**: Ermöglicht das Einzahlen eines bestimmten Betrags auf das Konto. Wirft eine IllegalArgumentException bei negativen Beträgen und eine IllegalStateException, wenn das Konto nicht aktiv ist.
- **withdraw(double amount)**: Ermöglicht das Abheben eines bestimmten Betrags vom Konto. Wirft eine IllegalArgumentException bei negativen Beträgen, eine Exception bei unzureichendem Kontostand und eine IllegalStateException, wenn das Konto nicht aktiv ist.

- **getBalance()**: Gibt den aktuellen Kontostand zurück.
- **changeState(AccountState newState)**: Ändert den Status des Kontos (z.B. ACTIVE, SUSPENDED, CLOSED).

4.2.3 Die BankApp-Klasse

Die ‘BankApp‘-Klasse stellt eine JavaFX-Anwendung dar, die es Nutzern ermöglicht, Transaktionen durchzuführen und den Kontostand anzuzeigen. Sie bietet grundlegende Benutzeroberflächenkomponenten und Ereignisbehandlungslogik. Die wichtigsten Funktionen sind:

- **start(Stage primaryStage)**: Initialisiert die JavaFX-Oberfläche mit Textfeldern und Buttons für Einzahlungen, Abhebungen und Zinsanwendungen.
- **setAccount(BankAccount newAccount)**: Setzt das aktuelle Konto auf ein neues BankAccount-Objekt.
- **formatBalance(double balance)**: Formatiert den Kontostand als String.

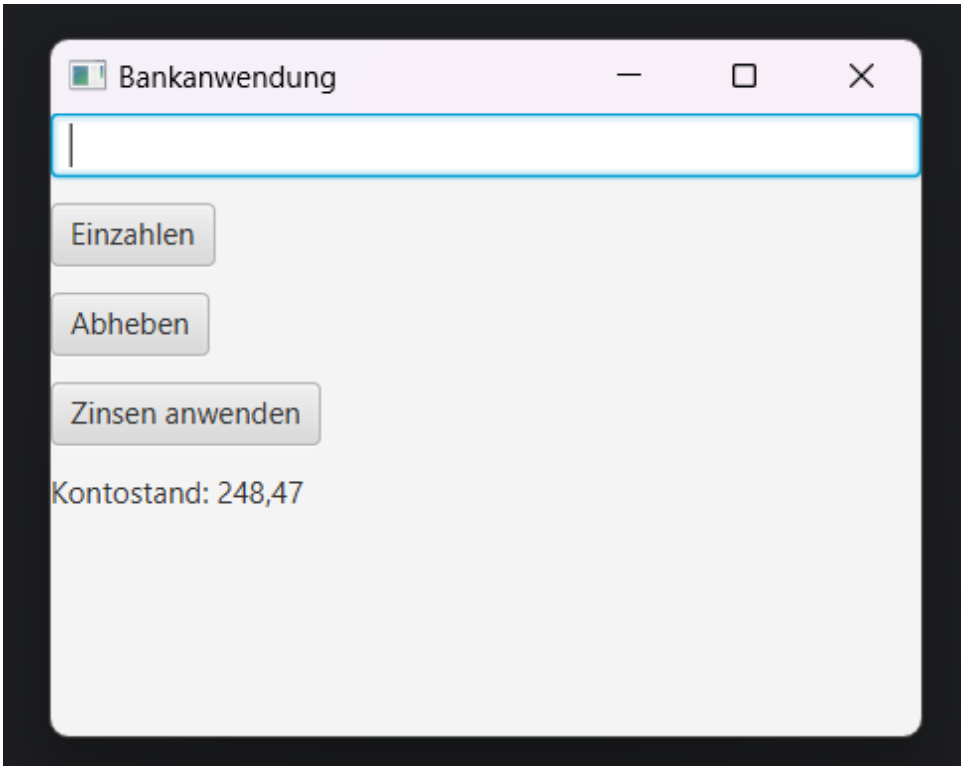

Abbildung 9: UI der Bank-Manager-App

4.2.4 Die BankManagings-Klasse

Die ‘BankManagings‘-Klasse enthält statische Methoden zur Berechnung von Gebühren, Zinsen und zur Durchführung von Überweisungen. Zu den wichtigsten Methoden gehören:

- **calculateFee(BankAccount account, double amount)**: Berechnet die Gebühr für eine Transaktion basierend auf dem Kontotyp.
- **calculateCompoundInterest(double principal, double rate, int times, int years)**: Berechnet die Zinsen mit Zinseszins.
- **transfer(BankAccount from, BankAccount to, double amount)**: Führt eine Überweisung von einem Konto zu einem anderen durch und berücksichtigt dabei die anfallenden Gebühren.
- **batchTransfer(List*BankAccount* accounts, BankAccount target, double amount)**: Führt Massenüberweisungen von mehreren Konten zu einem Zielkonto durch.

4.2.5 Die SavingsAccount-Klasse

Die ‘SavingsAccount‘-Klasse ist eine Unterklasse von ‘BankAccount‘ und fügt eine Methode zur Anwendung von Zinsen hinzu. Die wichtigste Methode ist:

- **applyInterest()**: Berechnet und fügt die Zinsen zum Kontostand hinzu.

Der vollständige Quellcode der vier Klassen: ‘BankAccount‘, ‘BankApp‘, ‘BankAccount‘, SavingsAccount‘ ist im **Anhang B.1 verfügbar.**

4.2.6 Generierung von Testklassen mit ChatGPT4

ChatGPT4 wird verwendet, um Testklassen für die oben beschriebenen Klassen zu generieren. Der Fokus liegt dabei auf der Überprüfung der Funktionalität der Methoden unter verschiedenen Bedingungen. Die generierten Testklassen umfassen Unit-Tests für die ‘BankAccount‘, ‘BankApp‘, ‘BankManagings‘ und ‘SavingsAccount‘-Klassen. Die Tests für JavaFX-Komponenten der ‘BankApp‘ sind einfach gehalten und dienen eher als Proof-of-Work, um die grundlegende Funktionsfähigkeit von ChatGPT4 zu demonstrieren.

Die Konversation mit ChatGPT für das generieren der Initialen Test-Klassen ist im **Anhang B.4 verfügbar.**

1. Generierte Testklasse für die BankAccount-Klasse
Die generierte Testklasse für die ‘BankAccount‘-Klasse überprüft Methoden wie ‘deposit‘ und ‘withdraw‘. Insbesondere wird geprüft, ob die Methoden korrekt mit verschiedenen Eingabewerten umgehen und wie sie auf ungültige Operationen reagieren. Diese Tests umfassen auch die Behandlung von Ausnahmen durch Try-Catch-Blöcke, die Verwendung von Enums zur Darstellung des Kontotyps und Kontostatus sowie die Implementierung eines einfachen Zustandautomaten.
Ein Beispiel für den Test der Methode ‘deposit‘, der auch die Ausnahmebehandlung überprüft:

Listing 5: Test für die Deposit-Methode mit Ausnahmebehandlung

```
@Test
void depositValidAmount() {
      account.deposit(100.0);
      assertEquals(100.0, account.getBalance());
}

@Test
void depositNegativeAmount() {
      Exception exception = assertThrows(
          IllegalArgumentException.class, () -> {
            account.deposit(-50.0);
      });
      assertEquals("Deposit␣amount␣must␣be␣positive.",
          exception.getMessage());
}
```

Dieses Beispiel wurde gewählt, um die Fähigkeit von ChatGPT zu demonstrieren, Tests zu generieren, die sowohl positive als auch negative Szenarien abdecken, einschließlich der richtigen Handhabung von Ausnahmen. Die erfolgreiche Abdeckung dieser Szenarien trägt erheblich zur Erhöhung der Methoden- und Branch-Coverage bei.

2. Generierte Testklasse für die BankApp-Klasse
Die generierte Testklasse für die ‘BankApp‘-Klasse enthält einfache GUI-Tests, die mit TestFX durchgeführt werden. Diese Tests simulieren grundlegende Benutzereingaben und überprüfen die Funktionalität der JavaFX-Komponenten. Diese Tests sind bewusst einfach gehalten, um die Fähigkeit von ChatGPT zu demonstrieren, grundlegende Testfälle für GUI-Anwendungen zu generieren.

Ein Beispiel für einen einfachen GUI-Test mit TestFX:

Listing 6: GUI-Test für die BankApp-Klasse mit TestFX

```
@Test
void testDepositInteraction() {
      clickOn("#amountField").write("500");
      clickOn("#depositButton");
      verifyThat("#balanceField", hasText("Kontostand:␣
          500,00"));
}
```

Dieses Beispiel wurde gewählt, um die Fähigkeit von ChatGPT zu demonstrieren, einfache Benutzerinteraktionen zu testen und sicherzustellen, dass die grundlegende Funktionalität der GUI-Komponenten korrekt implementiert ist. Obwohl diese Tests einfach sind, tragen sie zur Erhöhung der Methoden-Coverage bei, indem sie die grundlegenden Interaktionen abdecken.

3. Generierte Testklasse für die BankManagings-Klasse
Die generierte Testklasse für die 'BankManagings'-Klasse testet statische Methoden wie 'calculateFee'. Die Tests decken auch verschachtelte Schleifen und Switch-Case-Strukturen ab, was die Komplexität und Genauigkeit der generierten Tests demonstriert.
Ein Beispiel für den Test der Methode 'calculateFee', die eine Switch-Case-Struktur verwendet:

Listing 7: Test für die calculateFee-Methode mit Switch-Case

```
@Test
void calculateFeeChecking() {
      BankAccount account = new BankAccount("12345",
          BankAccount.AccountType.CHECKING);
      assertEquals(1.0, BankManagings.calculateFee(account,
          100.0));
}
```

Dieses Beispiel wurde gewählt, um zu zeigen, dass ChatGPT in der Lage ist, Kontrollstrukturen wie Switch-Case-Anweisungen korrekt zu testen und sicherzustellen, dass alle Fälle abgedeckt sind. Die Abdeckung solcher Strukturen trägt zur Erhöhung der Branch-Coverage bei.

4. Generierte Testklasse für die SavingsAccount-Klasse
Die generierte Testklasse für die ‘SavingsAccount‘-Klasse testet die spezifische Methode ‘applyInterest‘. Der Test stellt sicher, dass die Zinsen korrekt berechnet und dem Kontostand hinzugefügt werden. Die Java-Klasse nutzt Lambda-Ausdrücke zur Berechnung der Zinsen.

Ein Beispiel für den Test der Methode ‘applyInterest‘, die Lambda-Ausdrücke verwendet:

Listing 8: Test für die applyInterest-Methode mit Lambda-Ausdrücken

```
@Test
void applyInterest() {
      savingsAccount.deposit(1000.0);
      savingsAccount.applyInterest();
      assertEquals(1050.0, savingsAccount.getBalance(),
          0.01);
}
```

Dieses Beispiel wurde gewählt, um zu demonstrieren, dass ChatGPT in der Lage ist, komplexe Berechnungen und die Nutzung moderner Java-Features wie Lambda-Ausdrücke in Testfälle zu integrieren. Die erfolgreiche Abdeckung dieser Funktionalität trägt sowohl zur Methoden- als auch zur Branch-Coverage bei.

Der vollständige Quellcode der vier Test-Klassen: ‘BankAccount‘, ‘BankApp‘, ‘BankAccount‘, SavingsAccount‘ ist im **Anhang B.2 verfügbar.**

4.2.7 Analyse und Ergebnisse

A) Fehlerrate und Fehlerbehebung
Die initiale Testgenerierung durch ChatGPT4 zeigt einige Fehler, die durch die Assert-Ausgaben aufgedeckt werden. Bei der ‘BankAccount‘-Klasse gibt es zwei Rechenfehler, die zu falschen Assert-Ergebnissen führen. In der ‘BankApp‘-Klasse gibt es ein kleiner Syntaxfehler, sowie ein weiterer Rechenfehler. Diese Fehler werden aber durch Prompt-Engineering behoben.

B) Code-Coverage
Die initiale Testgenerierung durch ChatGPT4 liefert zufriedenstellende Ergebnisse hinsichtlich der Code-Coverage, allerdings gibt es noch einige ungetestete Bereiche:

- **BankAccount**: Einige Catch-Blöcke und geworfene Exceptions werden nicht vollständig getestet.
- **BankApp**: Einige Catch-Blöcke und die Methode ‘applyInterestInteraction‘ werden übersprungen.
- **BankManagings**: Ein Case und der Default im Switch-Case werden nicht getestet. Die verschachtelte Schleife wird ebenfalls übersprungen und bleibt ungetestet.

Vor dem Prompt-Engineering erreicht das System somit eine eine Methoden-Coverage von 84%, eine Branch-Coverage von 66%. Diese Abdeckung zeigt, dass grundlegende Funktionen gut getestet werden, jedoch spezifische Bereiche wie Catch-Blöcke und verschachtelte Schleifen nicht vollständig abgedeckt sind.

Nach gezieltem Prompt-Engineering und der Anpassung der generierten Testklassen wird die Code-Coverage deutlich verbessert. Dadurch werden die zuvor ungetesteten Bereiche abgedeckt und die Gesamtqualität der Tests gesteigert. Nach den Anpassungen erreicht das System eine vollständige Abdeckung in beiden Kategorien: 100% Methoden- und Branch-Coverage. Dies zeigt die Fähigkeit von ChatGPT4, durch gezieltes Feedback die Testabdeckung effektiv zu verbessern und somit die Code-Coverage zu steigern.

C) Aufwand für das Promptengineering
Der Aufwand für das Prompt-Engineering beim Erstellen der Testklassen ist wenig. Die initialen Prompts enthalten grundlegende Informationen zu den Klassen und deren Methoden. Durch Feedback und gezielte Anpassungen können die generierten Tests schrittweise verbessert werden. Dieser Verbesserungs-Prozess umfasst:

- Übermittlung der getesteten Klassen und der generierten Testklassen als Anhang im Prompt.
- Einfügen der in der IDE aufgetretenen Fehlermeldungen und eine kurze Erklärung der Fehlerursachen.
- Erwähnung der nicht getesteten Methoden und Branches im Prompt, um ChatGPT aufzufordern, diese in die Tests zu integrieren.

Dieser Prozess wird so lange wiederholt, bis die Tests fehlerfrei sind und die höchstmögliche Code-Coverage erreicht ist.

Für die **‘BankAccount‘**-Klasse generiert ChatGPT4 fast alle Tests korrekt. Eine spezifische Exception benötigt jedoch eine genaue Aufforderung, um eine vollständige Abdeckung zu erreichen.

Bei der **‘BankApp‘**-Klasse deckt ChatGPT4 nicht nur alle Testfälle ab, sondern schlägt auch vor, den Code zu modifizieren, um einen nicht testbaren Branch testbar zu machen. Dies beinhaltet das Hinzufügen eines Setters, um die vollständige Branch-Coverage zu erreichen.

In der **‘BankManagings‘**-Klasse werden seltene Verzweigungen wie ein ‘catch‘-Block oder das ‘default‘-Case im Switch erst nach wiederholter Aufforderung korrekt getestet. ChatGPT4 schlägt dabei auch Code-Änderungen vor, um die Testbarkeit zu verbessern, beispielsweise durch eine kleine Modifikation in einem Enum.

Dieser Prozess zeigt, dass ChatGPT4 nicht nur in der Lage ist, Testklassen zu generieren, sondern auch aktiv gute Vorschläge zur Verbesserung des Codes macht, um die Testbarkeit zu verbessern. ChatGPT4 kann durch gezielte Aufforderungen dazu gebracht werden, ungetestete Zweige und seltene Verzweigungen zu testen, was die Code-Coverage erheblich steigert. Außerdem ist es in der Lage, Fehler zu identifizieren und Lösungen vorzuschlagen, um diese zu beheben. Beispielsweise werden durch die Vorschläge von ChatGPT4 Setter-Methoden hinzugefügt oder kleine Code-Änderungen vorgenommen, um schwer testbare Zweige testbar zu machen. Insgesamt trägt dieser Prozess somit dazu bei, die Effizienz des Testentwicklungsprozesses zu verbessern.

D) Getestete Funktionalität
Die von ChatGPT4 generierten Testklassen decken eine Vielzahl von Funktionalitäten ab und zeigen eine hohe Zuverlässigkeit in der Testausführung. Die Tests umfassen folgende Funktionalitäten:

- **BankAccount**: Tests für ‘deposit‘, ‘withdraw‘, ‘getBalance‘ und ‘changeState‘, einschließlich Ausnahmebehandlung und Zustandautomaten.
- **BankApp**: Grundlegende GUI-Tests mit TestFX, die Benutzerinteraktionen wie Einzahlungen und Abhebungen simulieren.
- **BankManagings**: Tests für ‘calculateFee‘, ‘calculateCompoundInterest‘, ‘transfer‘ und ‘batchTransfer‘, einschließlich verschachtelter Schleifen und Switch-Case-Strukturen.

- **SavingsAccount**: Tests für ‘applyInterest‘, die Lambda-Ausdrücke zur Berechnung der Zinsen demonstrieren.

Die von ChatGPT4 erstellten Testklassen zeigen, dass sie die Hauptfunktionen der Java-Klassen zu sehr gut abdecken. Die Tests für Exception-Handling, verschachtelte Schleifen und Switch-Case-Strukturen demonstrieren die Fähigkeit von ChatGPT, komplexere Kontrollstrukturen korrekt und vollständig zu testen.

4.2.8 Fazit

Zusammenfassend zeigt sich für komplexere Testklassen und Unit-Tests, dass ChatGPT4 eine effiziente Unterstützung bei der Generierung von Testklassen für Java-Projekte bietet. Die vollständige Testabdeckung und die Fähigkeit zur schnellen Fehlerbehebung und Anpassung machen ChatGPT4 zu einem starken Werkzeug im Testentwicklungsprozess. Trotz kleinerer Schwächen kann ChatGPT4 durch gezieltes Prompt-Engineering hervorragende Ergebnisse erzielen und diese dienen als solide Grundlage für Unit-Tests, sowie UI-Tests.

4.3 Projekt 3: Legacy-Code Beispiel

4.3.1 Projektbeschreibung

In diesem Abschnitt werden für ein praxisnahes Projekt Testklassen mit ChatGPT4 generiert und näher untersucht. Das System besteht aus mehreren Packages und Klassen, die verschiedene Aspekte eines Verwaltungssystems für eine landwirtschaftliche Produktverwaltung repräsentieren. Zu den wichtigsten Packages gehören ‘app‘, ‘controllers‘, ‘data‘, ‘fileHandling‘, und ‘validation‘. Außerdem wird noch die UI mit CSS und FXML-Dateien erweitert.

Die Auswahl dieser Klassen basiert auf ihrer Fähigkeit, erweiterte Programmierkonzepte und Funktionalitäten zu demonstrieren, darunter Dateiverarbeitung, GUI-Interaktionen mit JavaFX, Validierung von Benutzereingaben und die Verwaltung komplexer Datenstrukturen. Diese Vielfalt an Programmierkonzepten macht sie zu einem idealen Kandidaten, um die Leistungsfähigkeit von ChatGPT4 bei der Generierung von Testklassen zu demonstrieren. Der Fokus dieses Projekts liegt auf komplexeren UI-Tests und Integrationstests.

4.3.2 Package: app

Dieses Package enthält die Hauptanwendungsklassen und die grundlegenden Utility-Klassen.

App-Klasse Die ‘App‘-Klasse ist die Hauptklasse der Anwendung und startet die JavaFX-Oberfläche.

Load-Klasse Die ‘Load‘-Klasse enthält Methoden zum Laden von FXML-Dateien und zur Verwaltung von Fenstern.

PathDialogBox-Klasse Die ‘PathDialogBox‘-Klasse bietet Dialoge zur Eingabe von Dateipfaden.

User-Klasse Die ‘User‘-Klasse repräsentiert einen Benutzer mit Benutzername und Passwort.

4.3.3 Package: controllers

Dieses Package enthält die Controller-Klassen für die verschiedenen FXML-Views.

AdminController-Klasse Die ‘AdminController‘-Klasse verwaltet die Hauptansicht für die Produktverwaltung.

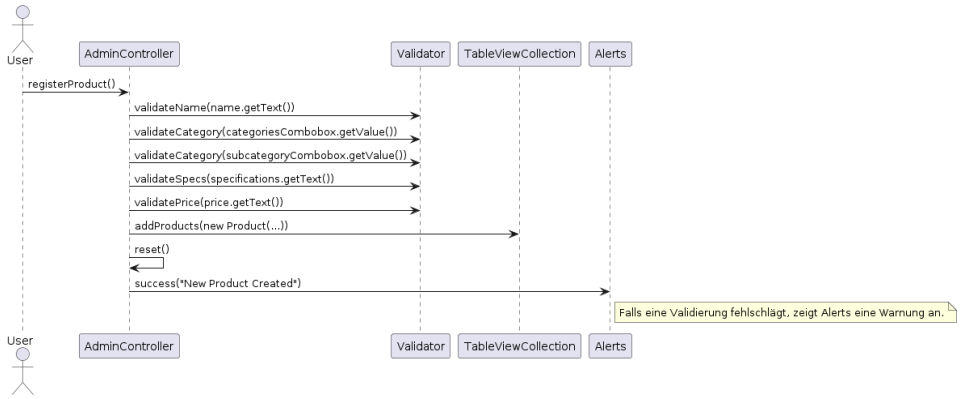

Abbildung 10: Sequenzdiagramm zum Hinzufügen eines Produkts

CategoryRegisterController-Klasse Die ‘CategoryRegisterController‘-Klasse verwaltet die Ansicht zum Hinzufügen neuer Kategorien und Unterkategorien.

LoginController-Klasse Die ‘LoginController‘-Klasse verwaltet die Login-Ansicht.

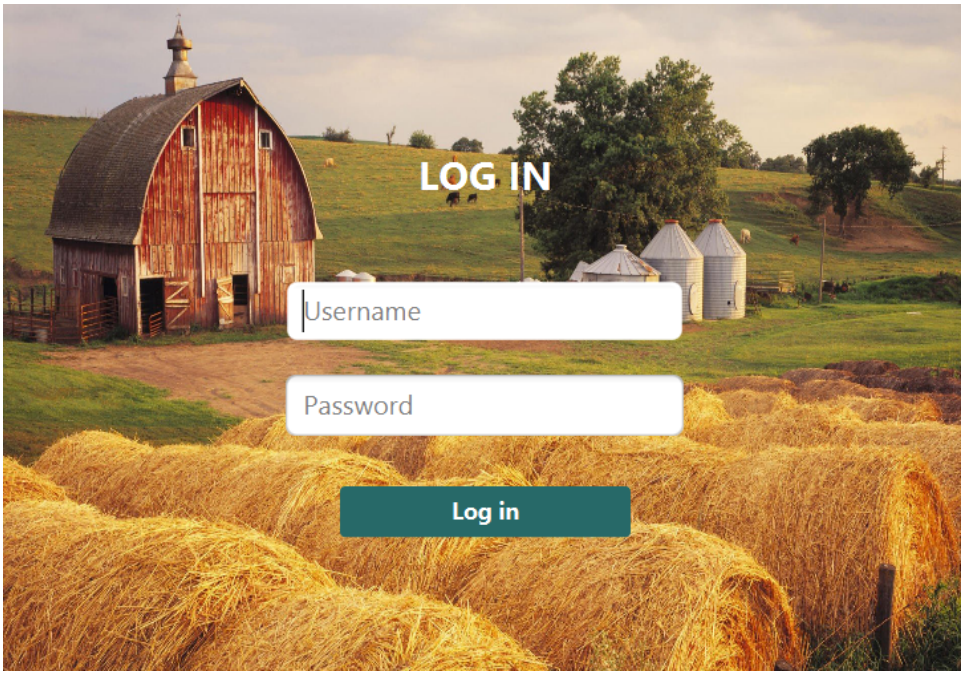

Abbildung 11: Login UI der Anwendung

4.3.4 Package: data

Dieses Package enthält die Datenmodelle und Datensammlungen.

Product-Klasse Die ‘Product‘-Klasse repräsentiert ein Produkt mit Attributen wie ID, Name, Kategorie, Spezifikationen und Preis.

Category-Klasse Die ‘Category‘-Klasse repräsentiert eine Produktkategorie mit einer Liste von Unterkategorien.

TableViewCollection-Klasse Die ‘TableViewCollection‘-Klasse verwaltet die Daten für die TableView und ermöglicht das Filtern, Sortieren und Bearbeiten von Produkten.

CategoryCollection-Klasse Die ‘CategoryCollection‘-Klasse verwaltet die Kategorien und deren Unterkategorien.

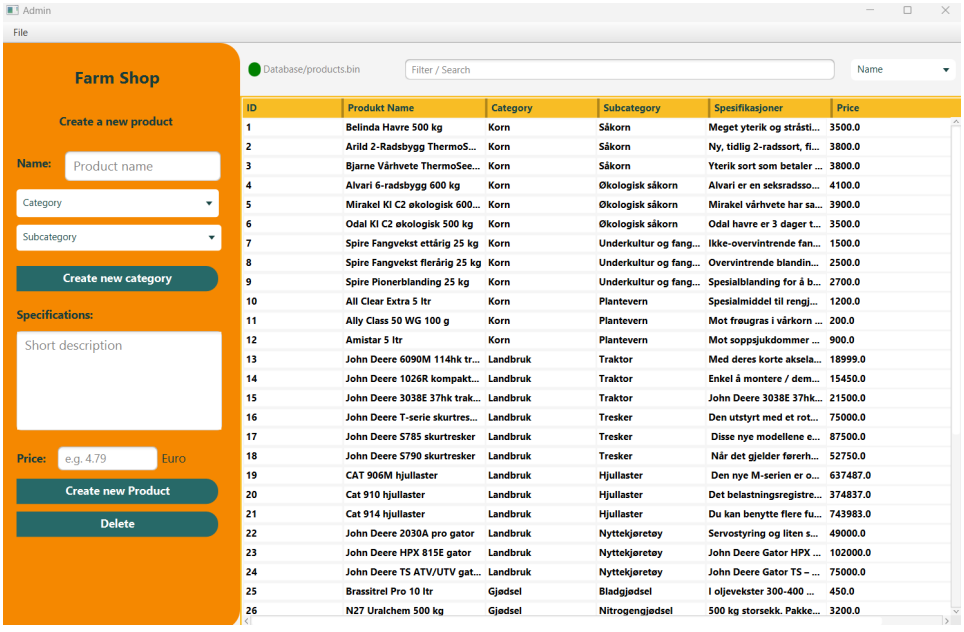

Abbildung 12: Haupt-UI der Anwendung

4.3.5 Package: fileHandling

Dieses Package enthält Klassen zur Handhabung von Dateioperationen.

FileInfo-Klasse Die ‘FileInfo‘-Klasse enthält Informationen zu Dateien, wie Pfad und Erweiterung.

IOClient-Klasse Die ‘IOClient‘-Klasse führt Dateioperationen wie das Speichern und Laden von Daten durch (siehe Abb. 13).

SaveThread-Klasse Die ‘SaveThread‘-Klasse führt Datei-Speicheroperationen in einem separaten Thread aus.

OpenThread-Klasse Die ‘OpenThread‘-Klasse führt Datei-Ladeoperationen in einem separaten Thread aus.

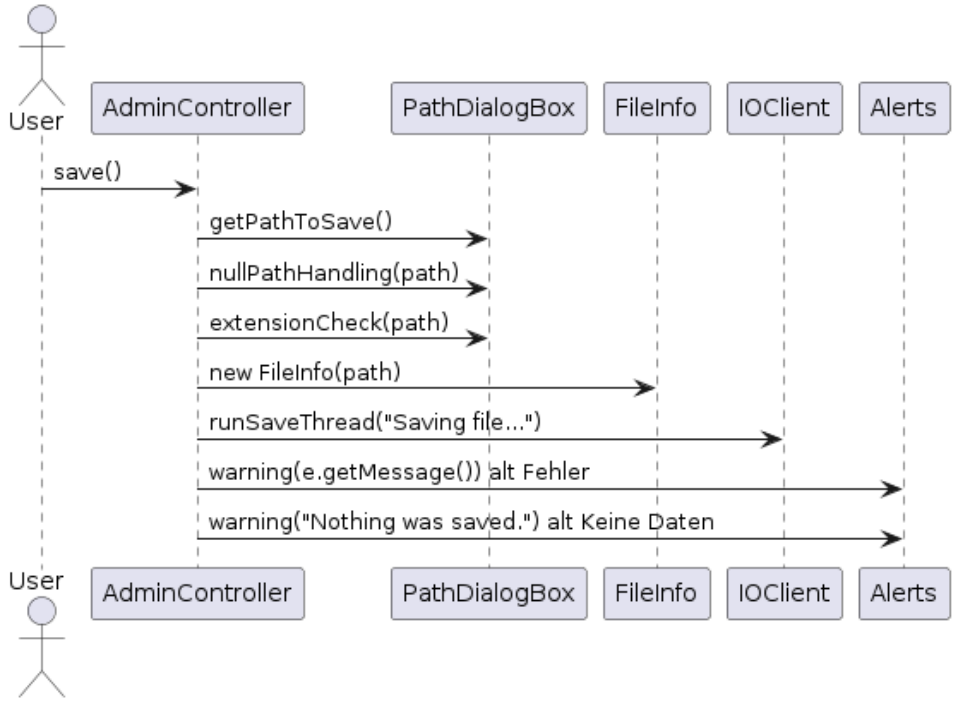

Abbildung 13: Sequenzdiagramm zum Speichern einer Datei

4.3.6 Package: validation

Dieses Package enthält Klassen zur Validierung von Benutzereingaben und zur Anzeige von Alerts.

Validator-Klasse Die ‘Validator‘-Klasse bietet Methoden zur Validierung von Eingabewerten wie IDs, Preisen und Textfeldern.

Alerts-Klasse Die ‘Alerts‘-Klasse zeigt verschiedene Arten von Alerts an, wie Warnungen und Bestätigungen.

NumberConversion-Klasse Die ‘NumberConversion‘-Klasse enthält Konverter für numerische Werte, die in der TableView verwendet werden.

Der vollständige Quellcode der Java-Klassen ist im **Anhang C.1 verfügbar.**

4.3.7 Generierung von Testklassen mit ChatGPT4

ChatGPT4 wird verwendet, um Testklassen für die oben beschriebenen Pakete und Klassen zu generieren. Der Fokus liegt dabei auf der Überprüfung der Funktionalität der Methoden unter verschiedenen Bedingungen. Die generierten Testklassen umfassen sowohl komplexe UI-Tests als auch Integrationstests, die die Interaktionen zwischen verschiedenen Komponenten der Anwendung überprüfen.

Die Konversation mit ChatGPT für das generieren der Initialen Test-Klassen ist im **Anhang C.4 verfügbar.**

1. Generierte Testklasse für die AdminController-Klasse
Die generierte Testklasse für die ‘AdminController‘-Klasse überprüft die Methoden zur Verwaltung der Hauptansicht für die Produktverwaltung. Hierzu gehören das Hinzufügen von Produkten und die Validierung der Benutzereingaben. Diese Tests tragen zur Erhöhung der Methoden- und Branch-Coverage bei, indem sie verschiedene Interaktionspfade abdecken.

Ein Beispiel für einen Integrationstest, der die Methode ‘registerProduct‘ testet:

Listing 9: Integrationstest für die registerProduct-Methode

```
@Test
void testRegisterProduct() {
      clickOn("#nameField").write("New␣Product");
      clickOn("#categoryField").write("Category1");
      clickOn("#subcategoryField").write("Subcategory1");
      clickOn("#priceField").write("19.99");
      clickOn("#addButton");
      verifyThat("#productTable", TableViewMatchers.hasTableCell("
          New␣Product"));
}
```

Dieses Beispiel zeigt, wie ChatGPT in der Lage ist, komplexe UI-Interaktionen zu testen, indem es die GUI-Elemente der Anwendung simuliert und die erwarteten Ergebnisse überprüft. Die erfolgreiche Abdeckung solcher Tests trägt zur Verbesserung der Methoden- und Branch-Coverage bei.

2. Generierte Testklasse für die LoginController-Klasse
Die generierte Testklasse für die 'LoginController'-Klasse testet die Login-Funktionalität der Anwendung. Diese Tests stellen sicher, dass die Benutzeranmeldungen korrekt verarbeitet werden und dass die Anwendung auf verschiedene Eingaben ordnungsgemäß reagiert.

Ein Beispiel für einen UI-Test mit TestFX, der die Login-Interaktionen überprüft:

Listing 10: UI-Test für die Login-Funktionalität

```
@Test
void testValidLogin() {
      clickOn("#usernameField").write("admin");
      clickOn("#passwordField").write("password123");
      clickOn("#loginButton");
      verifyThat("#welcomeMessage", hasText("Welcome, admin!"));
}

@Test
void testInvalidLogin() {
      clickOn("#usernameField").write("admin");
      clickOn("#passwordField").write("wrongpassword");
      clickOn("#loginButton");
      verifyThat("#errorMessage", isVisible());
}
```

Diese Beispiele demonstrieren die Fähigkeit von ChatGPT, Benutzereingaben zu simulieren und die entsprechenden Reaktionen der Anwendung zu überprüfen. Die Abdeckung von positiven und negativen Szenarien trägt zur Erhöhung der Methoden- und Branch-Coverage bei, indem sie verschiedene Pfade und Zustände der Login-Logik abdeckt.

3. Generierte Testklasse für die SaveThread-Klasse
Die generierte Testklasse für die 'SaveThread'-Klasse testet die Datei-Speicheroperationen in einem separaten Thread. Diese Tests sind entscheidend, um die korrekte Handhabung von Dateioperationen zu gewährleisten.

Ein Beispiel für einen Integrationstest, der die Methode ‘runSaveThread‘ testet:

Listing 11: Integrationstest für die runSaveThread-Methode

```
@Test
void testRunSaveThread() throws InterruptedException {
      SaveThread saveThread = new SaveThread("testfile.txt", data)
          ;
      saveThread.start();
      saveThread.join();
      File savedFile = new File("testfile.txt");
      assertTrue(savedFile.exists());
}
```

Dieses Beispiel zeigt, wie ChatGPT komplexe Dateioperationen und Multithreading-Mechanismen testet. Sie tragen zur Erhöhung der Methoden- und Branch-Coverage bei, indem sie sicherstellen, dass verschiedene Pfade und Zustände der Dateioperationen abgedeckt sind.

4. Generierte Testklasse für die Validator-Klasse
Die generierte Testklasse für die ‘Validator‘-Klasse testet die Methoden zur Validierung von Benutzereingaben. Diese Tests stellen sicher, dass die eingegebenen Daten den erwarteten Formaten und Bereichen entsprechen.

Ein Beispiel für einen Unit-Test, der die Methode ‘validatePrice‘ testet:

Listing 12: Unit-Test für die validatePrice-Methode

```
@Test
void testValidatePrice() {
      assertTrue(Validator.validatePrice("19.99"));
      assertFalse(Validator.validatePrice("invalid_price"));
      assertFalse(Validator.validatePrice("-10.00"));
}
```

Dieses Beispiel wurde gewählt, um zu zeigen, dass ChatGPT in der Lage ist, detaillierte Validierungslogik zu testen und sicherzustellen, dass Benutzereingaben korrekt verarbeitet werden. Die erfolgreiche Abdeckung dieser Tests trägt zur Verbesserung der Methoden- und Branch-Coverage bei.

Der vollständige Quellcode der Test-Klassen ist im **Anhang C.2 verfügbar.**

4.3.8 Analyse und Ergebnisse

A) Fehlerrate und Fehlerbehebung
Die initiale Generierung der Testklassen durch ChatGPT4 ergibt eine Vielzahl von Fehlern aufgrund der Komplexität der getesteten Funktionen. Trotz dieser Herausforderungen wird eine hohe Test-Coverage erreicht, auch wenn Teile, die Mocking benötigen, nicht enthalten sind. Die häufigsten Fehler umfassen ‘assertThrows‘ und ‘assertDoesNotThrow‘ Fehler sowie Initializer Errors in verschiedenen Klassen.

- **fileHandling**: Hier treten häufig Fehler in den Methoden auf, die Dateioperationen durchführen, insbesondere in den Test-Klassen ‘TestSaveCsvBehavior‘, ‘TestIOClient‘, ‘TestOpenThread‘, ‘TestSaveThread‘, ‘TestOpenAbstract‘ und ‘TestOpenCSVBehavior‘. Diese Fehler betreffen hauptsächlich die ‘assertThrows‘ und ‘assertDoesNotThrow‘ Assertions. Diese Fehler sind mit einfachem Promptengineering lösbar.
- **controllers**: Bei den Controller-Test-Klassen, insbesondere ‘TestAdminController‘, ‘TestCategoryRegisterController‘ und ‘TestLoginController‘, treten Initializer Errors auf, was zu einer 0% Coverage bei den Controllern führt. Es handelt sich hierbei um die UI-Teste.
- **data**: Die Datenmanagement-Klassen wie ‘TestCategoryCollection‘ und ‘TestTableViewCollection‘ haben fehlende Setup-Routinen und Initializer Errors. Zusätzliche Probleme treten bei einfachen Assert-Fehlern in ‘TestItemsFormat‘, ‘TestParseItems‘, ‘TestCategory‘ und ‘TestProduct‘ auf, diese sind aber sehr einfach durch Prompt-Engineering lösbar.
- **validation**: Die Validierungsklassen haben Initializer Errors sowie ‘assertThrows‘ und ‘assertDoesNotThrow‘ Fehler, insbesondere in den Klassen ‘Alerts‘ und ‘NumberConversion‘.

Hier noch eine Übersicht über die Initiale Testklassengenerierung. Insgesamt werden 27 Testklassen generiert, von denen:

- 8 Klassen keine Probleme aufweisen.
- 13 Klassen eine 100% Methoden-Coverage erreichen.
- 19 Klassen Probleme haben, die noch gelöst werden müssen.
- 8 Klassen ‘assertThrows‘ und/oder ‘assertDoesNotThrow‘ Fehler aufweisen.
- 4 Klassen Initializer Errors haben.
- 3 Klassen einfache Assert-Fehler zeigen.

- 2 Klassen eine 0% Code-Coverage aufweisen.

Die Gesamt-Methoden-Coverage beträgt somit in der Initialen Generierung 58%.

B) Code-Coverage
Die initiale Code-Coverage der generierten Tests durch ChatGPT4 ist nicht zufriedenstellend. Die Herausforderungen bei der Testgenerierung betreffen hauptsächlich die Komplexität der Validierungsklassen und das Fehlende Verständnis zur UI.

- **fileHandling**: Die initiale Code-Coverage in diesem Package ist durch die oben genannten Fehler eingeschränkt. Insbesondere Methoden, die Dateioperationen wie das Speichern und Laden von Daten in verschiedenen Formaten (CSV, Binär) ausführen, werden nicht vollständig abgedeckt. Dieses Problem ist aber schnell behoben durch Prompt-Engineering und die Methoden-Coverage für dieses Package steigt somit von 72% auf 86%.

- **controllers**: Bei den Controller-Klassen ist die Coverage durch Initializer Errors und fehlende Setup-Routinen begrenzt. Dies führt zu einer unvollständigen Abdeckung der Benutzerinteraktionen und UI-Logik. Durch sehr aufwendiges Prompt-Engineering, steigt die Methoden-Coverage in diesem Package von 0% auf 75%.

- **data**: Die Datenmanagement-Klassen zeigen eine Methoden-Coverage über dem Durchschnitt im Projekt, wobei einige Klassen wie ‘CategoryCollection‘ und ‘TableViewCollection‘ aufgrund von Setup-Problemen nicht vollständig getestet werden, dies ist aber durch einfaches Promptengineering lösbar. Außerdem haben Methoden zur Verwaltung von Kategorien und Produkten spezifische Validierungsfehler, die sehr genau beschrieben werden müssen im Prompt für eine Verbesserung. Diese Änderungen führen zu einem anstieg von 71% zu 95%, also fast vollständige, Methoden-Coverage.

- **Hauptklassen**: Die Coverage der Hauptklassen ist durch fehlendes Mocking eingeschränkt. Da es in diesem Projekt weniger über komplexes Mocking geht, wird für den Mocking erforderlichen Code-Teil keine Tests durchgeführt, welches sich auch in der Methoden-Coverage verdeutlicht. Methoden zur Benutzerverwaltung zeigen ‘assertDoesNotThrow‘ Fehler an, diese sind leicht zu beheben durch Promptengineering. Somit steigert sich die Methoden-Coverage von 50% auf 70%

- **validation**: Die Coverage der Validierungsklassen ist durch Initializer Errors und Assert-Fehler eingeschränkt. Methoden zur Validierung

von Benutzereingaben und Datenkonvertierung zeigen spezifische Fehler. Nach der Lösung von diesen Fehlern wird von 77% Methoden-Coverage, die 100% erreicht.

Durch gezieltes Prompt-Engineering und Anpassungen der generierten Tests können viele dieser Probleme behoben werden, was insgesamt zu einer verbesserten Code-Coverage der Tests führt. Diese Beträgt zusammengefasst eine Verbesserung von 58% auf 88% Methoden-Coverage.

C) Aufwand für das Promptengineering
Der Aufwand für das Promptengineering beim Erstellen der Testklassen ist im Vergleich zu den ersten beiden Beispielprojekten erheblich gestiegen. Dies liegt wahrscheinlich daran, dass praxisnahe Projekte wie dieses über Funktionen und Klassen verfügen, die weniger bekannte Strukturen aufweisen. Dadurch muss das Projekt im Prompt viel genauer beschrieben werden, einschließlich der Interaktionen zwischen den Komponenten und des genauen UI-Aufbaus. Klare und detaillierte Antworten auf diese Fragen sind notwendig, um ChatGPT präzise Anweisungen zu geben, was zu einem erheblichen Anstieg des Prompt-Engineering-Aufwands führt. Im Folgenden werden die spezifischen Herausforderungen und der Aufwand für das Promptengineering in den Packages ‘controllers‘ und ‘data‘ detailliert erläutert:

- **controllers**: Die Coverage wird zwar deutlich verbessert, aber UI-Elemente erfordern viel Promptengineering und detaillierte Beschreibungen. Warning-Fenster werden häufig in den Tests vergessen. ChatGPT erkennt und versteht die IDs in FXML-Dateien gut, doch die Testmethoden sind sehr fehleranfällig, wenn es darum geht, den UI-Aufbau korrekt zu interpretieren. Der UI-Aufbau muss sehr genau erklärt werden, um eine gute Testmethode mit hoher Test-Coverage zu erhalten. Meistens dient der generierte Code eher als Vorlage, die vom Softwareentwickler genauer angepasst werden muss, und ersetzt somit nicht, wie bei den einfachen Unit-Tests inm Projekt 1, den fast kompletten Softwareentwicklungsprozess durch einen Prompt-Engineering-Prozess.
- **data**: Die von ChatGPT generierte ‘TestParseItems‘-Klasse achtet nicht auf die Validator-Klasse, um Eingaben korrekt zu validieren. Die ‘Category‘- und ‘Product‘-Klassen sind zwar zu 100% testbar, weisen jedoch einfache Validierungsfehler auf, die von einem Softwareentwickler behoben werden müssen.

Eine Schwachstelle der generierten Tests ist jedoch die Validierung von Eingaben. ChatGPT erkennt oft nicht korrekt, wie die Validator-Klasse verwendet werden soll, was zu unvollständigen oder falschen Validierungen führt. Beispielsweise achtet die von ChatGPT generierte ‘TestParseItems‘-Klasse

nicht ausreichend auf die Validator-Klasse, um Eingaben korrekt zu validieren. Dies erfordert zusätzliche Anpassungen durch einen Softwareentwickler, um sicherzustellen, dass die Validierungen korrekt durchgeführt werden.

Zusammenfassend lässt sich sagen, dass nur durch umfangreiches Prompt-Engineering gute Tests für die UI generiert werden können. Integrationstests sind für ChatGPT einfacher zu erstellen als UI-Tests, da der Aufbau von Komponenten schneller verstanden wird. Es ist jedoch deutlich, dass ein Unterschied zu einfachen Unit-Tests besteht, die fast kein zusätzliches Prompt-Engineering erforderten.

D) Getestete Funktionalität
Die von ChatGPT4 generierten Testklassen decken eine Vielzahl von Funktionalitäten ab und zeigen eine hohe Zuverlässigkeit in der Testausführung. Die Tests umfassen folgende Funktionalitäten:

- **fileHandling**: Getestet werden die Klassen, die das Lesen und Schreiben von Daten in verschiedene Dateiformate wie CSV und Binär verwalten, einschließlich OpenThread, SaveThread, OpenBinBehavior und OpenCsvBehavior. Zusätzlich werden die Dateiinformationen durch die FileInfo-Klasse überprüft.
- **controllers**: Tests der Benutzerinteraktionen und UI-Logik, insbesondere der AdminController, LoginController und CategoryRegisterController, die verschiedene Aspekte der Benutzeroberfläche und -interaktionen steuern.
- **data**: Überprüfung der Datenmanagement-Klassen CategoryCollection und TableViewCollection, die das Hinzufügen, Entfernen und Laden von Datenobjekten verwalten. Außerdem werden die Datenmodelle Product und Category getestet, um sicherzustellen, dass die Daten korrekt gespeichert und verwaltet werden.
- **Hauptklassen**: Tests der App-Klasse für die Initialisierung der Anwendung und des Load-Mechanismus für das Laden und Wechseln von Ansichten sowie der User-Klasse, die die Benutzerdaten wie Benutzernamen und Passwörter behandelt.
- **customExeptions / ioExeptions**: Tests für benutzerdefinierte Ausnahmeklassen (EmptyFieldException, InvalidNumberFormat, InvalidTextInputException, FileDontExistsException, InvalidExtensionException, InvalidTypeException), die spezifische Fehlerzustände in der Anwendung behandeln.

- **validation**: Tests der Alerts-Klasse, die verschiedene Arten von Benachrichtigungen und Bestätigungsdialogen behandelt, der Number-Conversion-Klassen, die die Konvertierung von Zeichenketten in numerische Werte handhaben, und der Validator-Klasse, die sicherstellt, dass Eingabedaten den erwarteten Formaten und Regeln entsprechen.

Die Integrationstests spielen eine entscheidende Rolle bei der Überprüfung der Interaktionen zwischen verschiedenen Komponenten der Anwendung. Sie stellen sicher, dass die einzelnen Teile des Systems korrekt zusammenarbeiten. Beispielsweise überprüft der Integrationstest der 'AdminController'-Klasse nicht nur die Funktionalität des Produktregistrierungsprozesses, sondern auch die korrekte Integration mit den Datenmanagement- und Validierungskomponenten. Dies gilt ebenso für die 'LoginController'-Tests, die die Benutzeranmeldung und die damit verbundenen Benachrichtigungen und Validierungen sicherstellen.

Die von ChatGPT4 erstellten Testklassen zeigen, dass sie die Hauptfunktionen der Java-Klassen gut abdecken. Die Tests demonstrieren die Fähigkeit von ChatGPT, komplexere Kontrollstrukturen und Funktionalitäten korrekt und fast vollständig zu testen.

4.3.9 Fazit

Die Analyse und Ergebnisse der Testgenerierung durch ChatGPT4 für das praxisnahe Projekt zeigen sowohl Stärken als auch Schwächen auf. Die initiale Generierung der Testklassen erreicht eine nicht perfekte aber hohe Test-Coverage, obwohl sie durch verschiedene Fehler, insbesondere in komplexen Strukturen und bei der Validierung, eingeschränkt wird. Der Aufwand für das Promptengineering ist im Vergleich zu den ersten beiden Beispielprojekten deutlich höher, da detaillierte Beschreibungen und klare Anweisungen erforderlich sind, um zufriedenstellende Testergebnisse zu erzielen.

Während die Integrationstests weitgehend erfolgreich sind und die Zusammenarbeit zwischen den verschiedenen Komponenten der Anwendung gut überprüfen, erweisen sich die UI-Tests als besonders herausfordernd. Diese erfordern umfangreiches Promptengineering und detaillierte Beschreibungen, um eine hohe Test-Coverage zu erreichen. Die generierten Tests dienen hier eher als Vorlagen, die vom Softwareentwickler weiter angepasst werden müssen.

Insgesamt zeigt sich, dass ChatGPT4 in der Lage ist, eine Vielzahl von Funktionalitäten abzudecken und komplexe Kontrollstrukturen zu testen. Jedoch ist für praxisnahe Projekte ein erheblicher Aufwand an Promptengineering erforderlich, um die Qualität und Vollständigkeit der Tests sicherzustellen.

Die Unterschiede im Aufwand und in der Effektivität zwischen einfachen Unit-Tests, komplexeren Integrationstests und UI-Tests verdeutlichen die Notwendigkeit einer sorgfältigen und detaillierten Planung und Beschreibung der Testanforderungen.

5 Zusammenfassende Analyse und Vergleich der Projekte

5.1 Code-Coverage und Aufwand für das Promptengineering

Die Code-Coverage und der Aufwand für das Promptengineering variieren erheblich zwischen den drei Projekten. In den ersten beiden Projekten ist der Aufwand für das Promptengineering sehr gering und die erzielte Code-Coverage ist sehr hoch. Dies liegt daran, dass die Projekte einfache und klar definierte Strukturen haben, die ChatGPT leicht verstehen und testen kann. Im dritten Projekt, dem Legacy-Code-Beispiel, ist der Aufwand für das Promptengineering erheblich höher. Dies liegt daran, dass praxisnahe Projekte oft komplexe und weniger bekannte Strukturen enthalten, die detaillierte und präzise Beschreibungen im Prompt erfordern. Trotz dieser Herausforderungen kann durch umfangreiches Promptengineering eine zufriedenstellende Code-Coverage erreicht werden.

5.2 Unit-Tests

In allen drei Projekten zeigt sich, dass ChatGPT in der Lage ist, effektive Unit-Tests zu generieren. Die generierten Tests decken die Hauptfunktionen der Klassen gut ab und erreichen eine hohe Methoden-Coverage. Besonders in den ersten beiden Projekten kann durch minimalen Aufwand eine nahezu vollständige Abdeckung erzielt werden. Im dritten Projekt müssen die Prompts detaillierter gestaltet werden, um die gleichen Ergebnisse zu erzielen, aber letztlich werden auch hier solide Unit-Tests generiert.

5.3 UI-Tests

Die Erstellung von UI-Tests erweist sich als deutlich komplexer. In allen Projekten erfordert dies umfangreiches Promptengineering, insbesondere bei der Beschreibung des genauen UI-Aufbaus und der Interaktionen. Im dritten Projekt ist dies besonders ausgeprägt, da die generierten Tests häufig fehlende Elemente wie Warning-Fenster aufweisen und die Testmethoden fehleranfällig sind. Trotz dieser Herausforderungen kann durch gezieltes Promptengineering eine akzeptable Testabdeckung erreicht werden, wobei der generierte Code für die UI-Teste oft eher als Vorlage für weitere Anpassungen durch den Entwickler dient.

5.4 Integrationstests

Integrationstests stellen sich als weniger problematisch heraus als UI-Tests. In allen Projekten kann ChatGPT durch präzise Prompts Tests generieren, die die Interaktionen zwischen verschiedenen Komponenten der Anwendung abdecken. Im dritten Projekt zeigt sich jedoch, dass die generierten Tests

die Validierungsklassen nicht immer korrekt berücksichtigen, was zusätzliche Anpassungen durch den Entwickler erforderlich macht. Dennoch erreichen die Integrationstests insgesamt eine gute Abdeckung und tragen zur Sicherstellung der Systemintegrität bei.

5.5 Zusammenfassende Analyse der Projekte

Zusammenfassend lässt sich sagen, dass ChatGPT eine wertvolle Unterstützung bei der Generierung von Testklassen bietet. Besonders bei Unit-Tests kann durch minimalen Aufwand eine hohe Testabdeckung erzielt werden. Die Erstellung von UI-Tests erfordert hingegen erheblichen Promptengineering-Aufwand und detaillierte Beschreibungen, um eine akzeptable Abdeckung zu erreichen. Integrationstests erweisen sich als effektiver und weniger aufwendig als UI-Tests, können jedoch ebenfalls von präzisen Prompts profitieren.

Die Analyse der drei Projekte zeigt, dass ChatGPT in der Lage ist, komplexe Testanforderungen zu erfüllen, jedoch stark von der Qualität und Präzision der Prompts abhängt. Während einfache Unit-Tests nahezu automatisch generiert werden können, erfordern komplexere Testszenarien wie UI- und Integrationstests eine sorgfältige und detaillierte Vorbereitung der Prompts. Durch Anpassungen und Verbesserungen des Promptengineerings kann jedoch eine sehr gute Test-Coverage erreicht werden, die die Qualität und Zuverlässigkeit der Software deutlich verbessert.

6 Schlussfolgerungen und Ausblick

6.1 Zusammenfassung der wichtigsten Erkenntnisse

In dieser Arbeit wird die Fähigkeit von ChatGPT4 zur Generierung von Testklassen für Java-Projekte untersucht. Die wichtigsten Erkenntnisse umfassen:

- ChatGPT4 ist in der Lage, effektive Testklassen zu generieren, die eine hohe Methoden- und Branch-Coverage erzielen.
- Der Aufwand für das Prompt-Engineering variiert je nach Komplexität der zu testenden Klassen und Funktionen erheblich.
- Während einfache Unit-Tests mit minimalem Aufwand generiert werden können, erfordern komplexere UI- und Integrationstests detaillierte Anweisungen.
- Die generierten UI-Tests sind meistens nicht in der Lage, eine Vielzahl von UI-Elementen abzudecken, Grund ist das die UI-Struktur öfters dem Modell nicht richtig klar ist, wenn es nicht detailliert genug erklärt wird.

6.2 Diskussion der Effektivität von ChatGPT bei der Testgenerierung

Die Effektivität von ChatGPT4 bei der Generierung von Testfällen zeigt sowohl Vorteile als auch Nachteile im Vergleich zu traditionellen Methoden:

- **Vorteile:**
  - Schnelle Generierung von Testklassen, die eine hohe Abdeckung bieten.
  - Reduzierung des manuellen Aufwands durch Generierung.
  - Fähigkeit, komplexe Kontrollstrukturen und verschiedene Teststufen abzudecken.
- **Nachteile:**
  - Erhöhter Aufwand für das Prompt-Engineering bei komplexen Projekten.
  - Öfters besteht die Notwendigkeit der Überprüfung und Anpassung generierter Tests durch einen erfahrenen Entwickler.
  - Einschränkungen bei der Generierung von Tests für spezifische oder unbekannte Projekt-Strukturen.

6.3 Bewertung der Benutzerfreundlichkeit und Praxistauglichkeit

Die Anwendung von ChatGPT für die Testgenerierung zeigt, dass es sowohl praktische Vorteile als auch Herausforderungen gibt:

- **Benutzerfreundlichkeit:**
  - ChatGPT bietet eine benutzerfreundliche Oberfläche und ermöglicht es Entwicklern, schnell Testfälle zu generieren.
  - Die Notwendigkeit, präzise Prompts zu formulieren, kann jedoch eine Herausforderung darstellen, insbesondere für weniger erfahrene Entwickler.
- **Praxistauglichkeit:**
  - Die Integration von ChatGPT in den Entwicklungsprozess kann den Testaufwand erheblich reduzieren und die Effizienz steigern.
  - In komplexen Projekten erfordert die Anwendung von ChatGPT jedoch eine sorgfältige Planung und detaillierte Beschreibungen, um optimale Ergebnisse zu erzielen.
  - Die generierten Tests dienen oft als Ausgangspunkt und müssen vom Entwickler weiter angepasst werden, was die Praxistauglichkeit in bestimmten Szenarien einschränken kann.

6.4 Schlussbemerkungen

Die Untersuchung zeigt, dass ChatGPT4 ein vielversprechendes Werkzeug zur Unterstützung der Testgenerierung in Java-Projekten darstellt. Trotz einiger Herausforderungen und Einschränkungen bietet es bedeutende Vorteile, insbesondere bei der Reduzierung des manuellen Aufwands und der Verbesserung der Testabdeckung. Mit weiterer Forschung und Entwicklung könnte ChatGPT4 einen noch größeren Beitrag zur Automatisierung und Effizienzsteigerung im Softwaretestprozess leisten. Ein Ausblick auf die zukünftige Entwicklung und Anwendung von KI-gestütztem Testen ist daher sehr vielversprechend.

Literatur

[1] V. H. Guilherme und A. M. R. Vincenzi. “An initial investigation of ChatGPT unit test generation capability”. In: *8th Brazilian Symposium on Systematic and Automated Software Testing (SAST 2023)*. Sep. 2023.

[2] P. Liggesmeyer und S. Röttger. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. 2. Aufl. Springer Vieweg, 2014.

[3] Jasper Potts u. a. *JavaFX: Getting Started with JavaFX*. Release 8. E50607-02. Oracle. Aug. 2014. URL: http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

[4] A. Spillner und T. Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level - nach ISTQB®-Standard*. 6. Aufl. Heidelberg: dpunkt.verlag, 2019.

[5] JUnit Team. *JUnit 5 User Guide*. [Online]. Available: https://junit.org/junit5/docs/current/user-guide/ [Accessed: 16-Apr-2024]. JUnit. 2024.

[6] *Was ist Prompt Engineering?* Accessed: 2024-05-17. Amazon Web Services. 2024. URL: https://aws.amazon.com/de/what-is/prompt-engineering/.

A Anhang für InventoryManager

A.1 Java Klassen

Die Java-Klassen für das Projekt 1 sind im LaTeX-Projekt im Verzeichnis `implementation/A_project_1/01_java_classes` zu finden.

A.2 Von ChatGPT generierten Testklassen

Die von ChatGPT generierten Testklassen für das Projekt 1 sind im LaTeX-Projekt im Verzeichnis `implementation/A_project_1/02_generated_test_classes` zu finden.

A.3 Vollständiges Projekt

Das vollständige Projekt 1 ist im Verzeichnis `implementation/A_project_1/03_project_1` des LaTeX-Projekts zu finden oder auf GitHub: `https://github.com/EmirTutar/ItemManager`.

A.4 Konversation mit ChatGPT zur Testgenerierung

Die vollständige Konversation mit ChatGPT zur Testgenerierung für Projekt 1 ist unter folgendem Link verfügbar: `https://chat.openai.com/share/35f6fcf9-674b-4bb6-bc86-8f28478084b0`

B Anhang für BankManager

B.1 Java Klassen

Die Java-Klassen für das Projekt 2 sind im LaTeX-Projekt im Verzeichnis `implementation/B_project_2/01_java_classes` zu finden.

B.2 Von ChatGPT generierten Testklassen

Die von ChatGPT generierten Testklassen für das Projekt 2 sind im LaTeX-Projekt im Verzeichnis `implementation/B_project_2/02_generated_test_classes` zu finden.

B.3 Vollständiges Projekt

Das vollständige Projekt 2 ist im Verzeichnis `implementation/B_project_2/03_project_2` des LaTeX-Projekts zu finden oder auf GitHub: `https://github.com/EmirTutar/BankManager`.

B.4 Konversation mit ChatGPT zur Testgenerierung

Die vollständige Konversation mit ChatGPT zur Testgenerierung für Projekt 2 ist unter folgendem Link verfügbar: `https://chat.openai.com/share/0ba44806-082a-4c5b-8930-cfef8f3bab4e`

C Anhang für Farm-Product-Store

C.1 Java Klassen

Die Java-Klassen für das Projekt 3 sind im LaTeX-Projekt im Verzeichnis `implementation/C_project_3/01_java_classes` zu finden.

C.2 Von ChatGPT generierten Testklassen

Die von ChatGPT generierten Testklassen für das Projekt 3 sind im LaTeX-Projekt im Verzeichnis `implementation/C_project_3/02_generated_test_classes` zu finden.

C.3 Vollständiges Projekt

Das vollständige Projekt 3 ist im Verzeichnis `implementation/C_project_3/03_project_3` des LaTeX-Projekts zu finden oder auf GitHub: `https://github.com/EmirTutar/Farm-Product-Store`.

C.4 Konversation mit ChatGPT zur Testgenerierung

Die vollständige Konversation mit ChatGPT zur Testgenerierung für Projekt 3 ist unter folgendem Link verfügbar: `https://chatgpt.com/share/479bb82e-dc0b-4dc5-9318-a4dafa5f18cb`