WHZ Westsächsische Hochschule Zwickau
Hochschule für Mobilität

Westsächsische Hochschule Zwickau

Fakultät für Physikalische Technik und Informatik

Wissenschaftliche Arbeit

Thema:

Nutzung von Sprachmodellen zur Verbesserung von Bezeichnern in Java-Programmen

Vorgelegt von:	Kyz Saikal Tahirova
Matrikel-Nr.:	66110
Studiengang:	Informatik
Betreuer:	Prof. Dr. Laue
Abgabetermin:	xx.xx.2025

Inhaltsverzeichnis

Kapitel 1

1 Einleitung

In dieser Arbeit wird untersucht, wie große Sprachmodelle (LLMs) automatisch die Namen von Variablen, Methoden und Klassen in Java verbessern können. Verständliche Bezeichner erleichtern das Lesen und Warten von Code und sind entscheidend für die Softwarequalität [4]. In der Praxis treten jedoch häufig kurze, unklare oder konventionswidrige Namen auf, die die Lesbarkeit und Wartbarkeit einschränken.

Ein zentrales Problem sind sogenannte *linguistic anti-patterns* [1]: wiederkehrende sprachliche Muster in Bezeichnern, die zu Missverständnissen führen oder Java-Konventionen verletzen. Dazu gehören etwa generische Namen wie `data` oder `temp`, Abkürzungen, die schwer zu interpretieren sind, oder inkonsistente Schreibweisen. Die Arbeit untersucht, ob LLMs in der Lage sind, solche Fehler automatisch zu erkennen und bessere, verständliche Namen vorzuschlagen.

Ziel ist es, eine weitgehend automatische Lösung zu entwickeln:

1. Erstellung von Java-Beispielen mit schlechten Bezeichnern.
2. Übermittlung dieser Beispiele an ein Sprachmodell.
3. Automatische oder manuelle Auswertung der Vorschläge nach Verständlichkeit, Bedeutungsnähe und Einhaltung von Java-Konventionen.

Dabei wird insbesondere geprüft, welche Art von *Prompts* [3] am effektivsten ist, um konventionsgerechte und verständliche Bezeichner zu erzeugen und sprachliche Anti-Patterns zu vermeiden.

> **Forschungsfrage:** Wie gut können Sprachmodelle automatisch bessere Bezeichner in Java vorschlagen, ohne dass ein Mensch eingreifen muss?

Die Arbeit ist in mehrere Kapitel gegliedert:

1. **Kapitel 1 (Einleitung).** Vorstellung des Themas, Zielsetzung, Relevanz von Sprachmodellen in der Softwareentwicklung sowie die Forschungsfrage.
2. **Kapitel 2 (Theoretische Hintergrund).** Einführung in Java-Bezeichner, Namenskonventionen und linguistische Anti-Patterns.
3. **Kapitel 3 (Methodik und Durchführung).** Beschreibung der Vorgehensweise zur Untersuchung von Sprachmodellen, Erstellung von Java-Beispielen mit schlechten Bezeichnern und Auswertung der Ergebnisse. Eine automatische Pipeline wird nur theoretisch erwähnt.
4. **Kapitel 4 (Evaluation).** Die Ergebnisse zeigen, wie gut Sprachmodelle Vorschläge

hinsichtlich Verständlichkeit, Bedeutung, Konventionen und Vermeidung von Anti-Patterns liefern.

5. **Kapitel 5 (Diskussion und Schluss).** Grenzen der Methode, Verbesserungsideen, Ausblick auf weitere Forschung.

Hinweis: In dieser ersten Version der Arbeit wird der automatische Pipeline-Ansatz nur theoretisch erläutert, da die Umsetzung zum jetzigen Zeitpunkt noch nicht erfolgt ist. In einer späteren Version könnte diese Funktion ergänzt werden, um den gesamten Prozess der Bezeichnerverbesserung automatisch abzubilden.

Kapitel 2

2 Theoretische Hintergrund

2.1 Java-Bezeichner

Java-Bezeichner sind die Namen von Variablen, Methoden, Klassen, Schnittstellen, Konstanten und Paketen. Sie sollen die Funktion und den Zweck des Codes verständlich machen. Gute Bezeichner sind selbsterklärend, konsistent und folgen klaren Regeln [4]. Dies erleichtert das Lesen, Verstehen und Warten von Software.

2.2 Namenskonventionen in Java

Java verwendet für Bezeichner verschiedene Schreibweisen, die hauptsächlich auf CamelCase basieren. Dabei gilt:

- Klassen und Schnittstellen. Namen sind Substantive, jede Wortanfangsbuchstabe wird großgeschrieben, z.B. `Student`, `Scanner`, `Runnable`. Abkürzungen und Akronyme sollten vermieden werden.
- Methoden. Namen sind Verben, beginnen mit einem Kleinbuchstaben, innere Wortanfänge werden großgeschrieben, z.B. `calculateSum()`, `main()`.
- Variablen. Namen sollten kurz, aber aussagekräftig sein. Keine Unterstriche oder Dollarzeichen am Anfang. Ein-Zeichen-Variablen nur temporär (z.B. `i, j, k`). Beispiel: `totalScore`, `marks`.
- Konstanten. Alle Buchstaben groß, Wörter durch Unterstriche getrennt, z.B. `MAX_SIZE`, `PI_VALUE`.
- Pakete. Alles klein, oft in Anlehnung an Top-Level-Domains wie `com`, `org`. Beispiel: `java.util.Scanner`.

Die konsequente Anwendung dieser Regeln erhöht die Verständlichkeit und Wartbarkeit von Code, insbesondere in größeren Projekten.

2.3 Linguistische Anti-Patterns

Linguistic Anti-Patterns (LAs) beschreiben wiederkehrende schlechte Praktiken in der Benennung, Dokumentation und Implementierung von Softwareelementen. Sie stellen also sprachliche Widersprüche oder Inkonsistenzen zwischen Namen, Kommentaren und tatsächlichem Verhalten einer Methode oder eines Attributs dar [1].

Im Gegensatz zu klassischen Design-Anti-Patterns, die strukturelle Probleme betreffen, beziehen sich LAs auf die sprachliche Ebene des Codes. Sie zeigen sich zum Beispiel, wenn

eine Methode `get()` heißt, aber keinen Wert zurückgibt, oder eine Methode `isValid()` keinen booleschen Wert liefert. Ebenso gehören widersprüchliche Kommentare oder unklare Attributnamen zu häufigen Fällen solcher sprachlichen Inkonsistenzen.

Arnaoudova et al. (2013) [2] analysierten mehrere Open-Source-Projekte wie ArgoUML, Cocoon und Eclipse, um diese Muster systematisch zu erfassen. Dabei wurden typische Kategorien von LAs identifiziert:

- Inkonsistenz zwischen Name und Rückgabewert.
- Widersprüche zwischen Kommentar und Implementierung
- Mehrdeutige oder irreführende Bezeichner.
- Verwendung generischer Namen ohne semantischen Bezug.

Das Bewusstsein für linguistische Anti-Patterns hilft, Missverständnisse zu vermeiden und die Lesbarkeit, Verständlichkeit und Wartbarkeit von Software nachhaltig zu verbessern.

2.4 Große Sprachmodelle (LLMs)

Große Sprachmodelle (LLM), wie GPT oder CodeBERT, haben den Bereich der automatischen Programmierung grundlegend verändert. Sie ermöglichen nicht nur die Code-Erstellung in natürlicher Sprache, sondern führen auch zunehmend Aufgaben wie die Code-Erklärung, die Testgenerierung oder die Fehlerbehebung durch. Daher verlagert sich der Schwerpunkt der Softwareentwicklung von der manuellen Implementierung auf die Zusammenarbeit zwischen Mensch und Modell.

Wie in *Automatic Programming: Large Language Models and Beyond* [5] beschrieben, besteht jedoch weiterhin ein zentrales Problem in der Vertrauenswürdigkeit automatisch generierten Codes. Obwohl LLMs beeindruckende Ergebnisse liefern, bleibt die Frage offen, wann und unter welchen Bedingungen dieser Code als zuverlässig genug gilt, um in reale Projekte integriert zu werden. Neben der Korrektheit spielen auch Aspekte wie Sicherheit, Nachvollziehbarkeit und rechtliche Verantwortung eine Rolle.

In dieser Arbeit wird untersucht, inwiefern Sprachmodelle diese Aufgaben zuverlässig ausführen und in welchen Fällen menschliches Eingreifen weiterhin notwendig bleibt.

Kapitel 3

3 Methodik und Durchführung

Dieses Kapitel erklärt die Auswahl der Sprachmodelle, die Erstellung von Test-Beispielen nach den Kategorien von Arnaoudova et al. [1] und die Bewertungskriterien. Das Ziel ist, systematisch zu prüfen, wie gut LLMs linguistische Anti-Patterns in Java-Code erkennen und verbessern können.

3.1 Auswahl der Sprachmodelle

Für die Untersuchung wurden drei verschiedene Modelle gewählt:

- **ChatGPT-4 (OpenAI).** Ein sehr leistungsstarkes kommerzielles Modell mit ausgezeichneten Programmierfähigkeiten. Es versteht natürliche Sprache sehr gut und kann komplexen Code analysieren.
- **Hawki (WHZ).** Das KI-System der Westsächsischen Hochschule Zwickau. Eine lokale Lösung, die für akademische Zwecke entwickelt wurde.
- **Claude 3.5 OPUS (Anthropic).** Bekannt für präzise Antworten und gute Befolgung von Anweisungen. Liefert strukturierte und durchdachte Ausgaben.

Die Auswahl ermöglicht den Vergleich zwischen verschiedenen kommerziellen Lösungen und einem akademischen System.

3.2 Erstellung der Test-Beispiele

3.2.1 Kategorien nach Arnaoudova

Die Kategorien wurden in zwei Hauptgruppen unterteilt:
Methoden-bezogene Anti-Patterns (A, B, C):

1. **Kategorie A** (*does more than it says*) – die Methode macht mehr, als der Name verspricht.
   - **A1:** Methode mit `get`, die mehr macht als nur zurückgeben
   - **A2:** Methodenname ist Prädikat, aber Rückgabetyp ist nicht Boolean
   - **A3:** Methode mit `set`, die einen Wert zurückgibt
   - **A4:** Methodentyp deutet auf mehrere Objekte, Name auf einzelnes
2. **Kategorie B** (*says more than it does*) – der Name verspricht mehr, als die Methode tut.
   - **B1:** Kommentar dokumentiert nicht implementierte Bedingung

- **B2:** Validierungsmethode gibt nichts zurück
- **B3:** Name deutet Rückgabe an, aber Rückgabetyp ist `void`
- **B4:** Name ist Prädikat, aber nichts wird zurückgegeben

3. **Kategorie C** (*does the opposite*) – die Methode macht das Gegenteil vom Namen.
   - **C1:** Name und Rückgabetyp verwenden Antonyme (Gegensätze)
   - **C2:** Kommentar und Signatur verwenden Antonyme

Attribut-bezogene Anti-Patterns (D, E, F):

4. **Kategorie D** (*contains more than it says*) – das Attribut enthält mehr, als der Name aussagt.
   - **D1:** Attributname deutet Boolean an, aber Typ ist anders
   - **D2:** Attributname ist Prädikat, aber Typ ist nicht Boolean
5. **Kategorie E** (*says more than it contains*) – der Name sagt mehr, als das Attribut enthält.
   - **E1:** Attributtyp deutet auf einzelnes Objekt, Name auf mehrere
6. **Kategorie F** (*says the opposite*) – Name und Inhalt sind gegensätzlich.
   - **F1:** Attributname und Typ verwenden Antonyme
   - **F2:** Attributkommentar und Signatur verwenden Antonyme

3.2.2 Beispiel-Code

Kategorie A1. Methode mit `get` macht mehr als zurückgeben

```
public class UserManager {
    private User currentUser;

    // Problem: getUser() macht mehr als nur zuruckgeben
    public User getUser() {
        validateSession();      // Extra Aktion!
        logAccess();            // Extra Aktion!
        return currentUser;
    }
}
```

Problem: Eine `get`-Methode soll nur einen Wert zurückgeben, aber hier passiert mehr. Erwartete Verbesserung:

- Entweder umbenennen zu `fetchAndValidateUser()`
- Oder die Extra-Funktionen entfernen

Kategorie B3. Name deutet Rückgabe an, aber `void`:

```java
public class Calculator {
    private int result;

    // Problem: getName() gibt nichts zuruck (void)
    public void getName() {
        System.out.println("Calculator");
    }
}
```

Problem: `getName()` suggeriert einen Rückgabewert, aber der Typ ist `void`. Erwartete Verbesserung:

- Entweder `String getName()` mit `return "Calculator";`
- Oder umbenennen zu `printName()`

Kategorie C1. Name und Typ sind Gegensätze:

```java
public class Config {
    // Problem: disable() gibt ControlEnableState zuruck
    public ControlEnableState disable() {
        return new ControlEnableState(true);
    }
}
```

Problem: `disable()` gibt einen ËnableStatus zurück – Widerspruch! Erwartete Verbesserung:

- Entweder `ControlDisableState` zurückgeben
- Oder Methode zu `enable()` umbenennen

Insgesamt wurden 30 solcher Beispiele erstellt, je 5 für jede Kategorie A bis F.

3.3 Prompt-Strategien

Verschiedene Arten von Prompts wurden getestet, um herauszufinden, welche am besten funktionieren.

3.3.1 Zero-Shot Prompt (ohne Beispiele)

```
Verbessern Sie die Bezeichner im folgenden Java-Code:
[CODE EINFÜGEN]
```

Dieser Prompt gibt dem Modell minimale Informationen. Er testet, ob das Modell schon von sich aus gute Konventionen kennt.

3.3.2 Few-Shot Prompt (mit Beispielen)

```
Beispiele für gute Java-Namen:
1) Klassen: CustomerAccount, PaymentProcessor
2) Methoden: calculateTotal(), validateInput()
3) Variablen: userName, itemCount
Verbessere nun diesen Code:
[CODE EINFÜGEN]
```

3.3.3 Detaillierter Kontext-Prompt (mit ausführlichen Anweisungen)

```
Du bist ein erfahrener Java-Entwickler. Verbessere die Bezeichner:
1) Folge Java-Konventionen (CamelCase)
2) Vermeide linguistische Anti-Patterns
3) Wähle selbsterklärende Namen
4) Achte auf Kontext und Bedeutung
Code:
[CODE EINFÜGEN]
```

3.4 Durchführung der Tests

Das ergibt insgesamt 270 Tests (30 Beispiele $\times$ 3 Prompts $\times$ 3 Modelle).

3.4.1 Ablauf

1. Code-Beispiel mit linguistischem Anti-Pattern vorbereiten.
2. Kategorie notieren.
3. Prompt wählen (Zero-Shot, Few-Shot oder Detailliert).
4. An LLM senden.
5. Antwort aufzeichnen.
6. Bewerten nach 5 Kriterien.

3.5 Bewertungskriterien

Jedes Kriterium wird mit 1-5 Punkten bewertet.

1. **Anti-Pattern-Erkennung.** Wurde das Problem erkannt?
2. **Korrekturqualität.** Ist der vorgeschlagene neue Name besser? Löst er das Anti-Pattern?
3. **Konventions-Konformität:** Folgt der neue Code Java-Standards?

4. **Semantische Klarheit:** Sind die neuen Namen verständlich?
5. **Konsistenz:** Passt alles zusammen?

3.5.1 Spezielle Bewertung nach Kategorien

Für jede Kategorie A-F wurden spezifische Fragen gestellt:

- **Kategorie A:** Wurde die Extra-Funktionalität dokumentiert oder die Methode umbenannt?
- **Kategorie B:** Wurde der fehlende Rückgabewert hinzugefügt oder der Name korrigiert?
- **Kategorie C:** Wurden die widersprüchlichen Begriffe harmonisiert?
- **Kategorie D-F:** Wurden Attributnamen und Typen in Einklang gebracht?

Literatur

[1] Arnaoudova, V.; Di Penta, M.; Antoniol, G. (2014): *Linguistic Anti-Patterns: What They Are and How Developers Perceive Them.* Empirical Software Engineering. Verfügbar unter: veneraarnaoudova.ca.

[2] Conference: European Conference on Software Maintenance and Reengineering (2013): *A New Family of Software Anti-Patterns: Linguistic Anti-Patterns.* Verfügbar unter: assets.ptidej.net

[3] Startup Creator (2023): *Die besten ChatGPT Prompts.* Verfügbar unter: startup-creator.com.

[4] Oracle: *Naming Conventions.* Verfügbar unter: Java Naming Conventions.

[5] MICHAEL R., LYU; BAISHAKHI, R.; ABHIK, R.; SHIN HWEI TAN; PATANAMON, T.: *Automatic Programming: Large Language Models and Beyond.* Verfügbar unter: dl.acm.org