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

Sprachmodelle werden in der Softwareentwicklung immer häufiger eingesetzt, um Entwickler bei Analyse- und Review-Aufgaben zu unterstützen. Ein wichtiger Anwendungsbereich ist die Bewertung von Bezeichnern im Quellcode. Klare und verständliche Namen vereinfachen die Wartung, während verschwommene Bezeichner zu Fehlern und erhöhtem Aufwand führen [4].

Diese Arbeit untersucht die Zuverlässigkeit moderner Sprachmodelle bei der Einschätzung von Java-Bezeichnern. Bewertet werden Verständlichkeit, semantische Angemessenheit, Einhaltung etablierter Namenskonventionen [2] sowie die Erkennung linguistischer Anti-Muster [1]. Unter linguistischen Anti-Mustern versteht man systematische sprachliche Probleme in Bezeichnern, die Klarheit und Lesbarkeit beeinträchtigen.

Ziel der Untersuchung ist es zu prüfen, ob Sprachmodelle hilfreiche und konsistente Hinweise zur Verbesserung von Bezeichnern liefern können. Hierfür werden mehrere Prompt-Varianten entwickelt und schrittweise verfeinert. Die Analyse erfolgt anhand von Java-Beispielen mit sowohl problematischen als auch korrekten Bezeichnern, um die Erkennung tatsächlicher Verstöße und das Auftreten unbegründeter Warnungen („False Positives“) zu bewerten. Frühere Arbeiten zeigen bereits, dass LLM-generierter Code zu semantischen Fehlbenennungen und strukturellen Qualitätsproblemen neigt [5, 7, 3].

Die zentrale Forschungsfrage lautet:

> Wie zuverlässig können Sprachmodelle Java-Bezeichner hinsichtlich Verständlichkeit, Bedeutung, Konventionen und linguistischer Anti-Muster bewerten?

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).** Überblick über Java-Bezeichner, Namenskonventionen und linguistische Anti-Muster.
3. **Kapitel 3 (Methodik und Durchführung).** Aufbau der Untersuchung, Entwicklung der Prompt-Varianten sowie Beschreibung der verwendeten Java-Beispiele. *Eine automatische Pipeline wird nur theoretisch erwähnt.*
4. **Kapitel 4 (Evaluation).** Auswertung der Ergebnisse und systematische Bewertung der Antworten der Sprachmodelle.
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

Dieser Abschnitt bietet einen Überblick über Java-Bezeichner, etablierte Namenskonventionen sowie linguistische Anti-Muster, die als Grundlage für die spätere Analyse dienen. Die Relevanz dieser Konzepte ergibt sich aus empirischen Studien, die zeigen, dass Bezeichner einen erheblichen Einfluss auf die Verständlichkeit und Qualität von Quellcode haben [4].

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. Die Studie von Butler et al. [4] zeigt, dass unklare, inkonsistente oder mehrdeutige Namen mit geringer Wartbarkeit, erhöhter Komplexität und häufigeren Warnungen in statischen Analysewerkzeugen verbunden sind. Im Gegensatz dazu können klare Namen ein einfacher Indikator für eine hohe Qualität des Quellcodes sein.

2.2 Namenskonventionen in Java

Die offiziellen Java-Konventionen [2] empfehlen konsistente Schreibweisen und semantisch präzise Nomenphrasen. Dazu gehören:

- **Klassen und Schnittstellen.** Großschreibung jedes Wortanfangs (CamelCase), z. B. `CustomerAccount`, `PaymentProcessor`. Allgemein sollten aussagekräftige Nomenphrasen verwendet werden, etwa `EnrolledStudents` oder `NumberOfValidCreditCards`.
- **Methoden.** Verben oder Verbphrasen wie `calculateTotal()` oder `validateInput()`.
- **Attribute/Variablen.** camelCase und präzise Bedeutung, z. B. `userName`, `itemCount`.

Ein Name soll beschreiben, was ein Element speichert oder ausführt. Unklare Begriffe wie `tmp`, `data` oder `handle` lassen zu viel Interpretationsspielraum und führen schnell zu Missverständnissen.

2.3 Linguistische Anti-Muster

Linguistische Anti-Muster (Linguistic Anti-Patterns) wurden erstmals systematisch von Arnaoudova et al. beschrieben [1]. Sie bezeichnen wiederkehrende sprachliche Probleme in Bezeichnern, Kommentaren oder Signaturen, bei denen Name und tatsächliches Verhalten nicht übereinstimmen. Dies führt zu kognitiven Brüchen und erschwert das

Programmverständnis.

Die Anti-Muster lassen sich in zwei Gruppen einteilen.

Methodenbezogene Anti-Muster:

1. **Kategorie A: „tut mehr als der Name sagt“** Die Methode führt zusätzliche Aktionen aus. Beispiel: `getUser()` validiert oder speichert Daten.
2. **Kategorie B: „sagt mehr als sie tut“** Der Name verspricht Funktionalität, die nicht implementiert ist. Beispiel: `validate()` besitzt den Rückgabetyp `void`.
3. **Kategorie C: „tut das Gegenteil“** Name und Verhalten widersprechen sich. Beispiel: `disable()` erzeugt einen `EnableState`.

Attributbezogene Anti-Muster:

4. **Kategorie D: enthält mehr als der Name sagt** Beispiel: ein als Prädikat benannter Bezeichner, dessen Typ jedoch kein `boolean` ist.
5. **Kategorie E: Name sagt mehr als enthalten ist** Beispiel: `users` als Name für ein einzelnes Objekt.
6. **Kategorie F: Name und Inhalt widersprechen sich** Beispiel: Attributname und Typ bilden Antonyme.

2.4 Große Sprachmodelle (LLMs)

Sprachmodelle werden zunehmend zur Bewertung von Quellcode verwendet [3]. Studien zeigen jedoch, dass LLMs zwar Muster erkennen können, aber gleichzeitig zu einer oberflächlichen Analyse neigen, insbesondere wenn nur Identifikatoren geändert oder semantisch irrelevante Änderungen vorgenommen werden [6]. Darüber hinaus zeigen mehrere Studien eine hohe Rate an Fehlalarmen („False Positives”) und ungenauen Bewertungen [5, 8].

In dieser Arbeit wird untersucht, wie zuverlässig Sprachmodelle Java-Bezeichner bewerten können und in welchen Fällen weiterhin menschliches Eingreifen erforderlich ist.

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 über HAWKI[1] abgefragt:

- OpenAI GPT-4o,
- Meta LLaMA 3.1 70B Instruct,
- Alibaba Qwen 2.5 72B Instruct.

3.2 Erstellung der Test-Beispiele

3.2.1 Kategorien nach Arnaoudova

Die linguistischen Anti-Muster nach Arnaoudova et al. [1] werden in zwei Gruppen unterteilt: methodenbezogene und attributbezogene Muster. Jede Kategorie markiert ein typisches semantisches Problem zwischen Bezeichner und Implementierung. Jedes Muster wurde kurz erklärt und durch ein Beispiel veranschaulicht. Insgesamt wurden 10–15 fehlerhafte Codebeispiele und 5–7 korrekte Beispiele erstellt.

Methoden-bezogene Anti-Muster (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

[1] HAWKI dient als Schnittstelle zu OpenAI GPT-4o, Meta LLaMA 3.1 70B Instruct, Alibaba Qwen 2.5 72B Instruct Modellen

- **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;

    public User getUser() {
        validateSession();   // Nebenwirkung
        return currentUser;
    }
}
```

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

```
public class Calculator {
    public void getResult() {
        System.out.println("42");
    }
}
```

Kategorie C1. Name und Typ sind Gegensätze:

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

Kategorie D2. Prädikatname, Typ nicht Boolean:

```
public class Payment {
    private int isActive; // Typ passt nicht zum Prädikat
}
```

Kategorie E1. Pluralname bei singular Typ:

```
public class Order {
    private Product products; // Name suggeriert Mehrzahl
}
```

Kategorie F1. Name und Typ im Gegensatz:

```
public class Metrics {
    private FailureCounter success; // Erfolg vs. Fehler
}
```

3.3 Prompt-Strategien und Iteration

Die Prompts wurden iterativ weiterentwickelt (V1 → V2 → V3 → V4). Ziel jeder Iteration war die Verbesserung der Hinweisqualität (automatisches Review) und die Reduzierung von False Positives. V4 enthält zudem eine knappe Definition der linguistischen Anti-Muster, damit Modelle diese Kategorien gezielt erkennen.

V1 (Zero-Shot) „Analysieren Sie den folgenden Java-Code. Identifizieren Sie mögliche Benennungs- oder Struktur-Anti-Muster.“

V2 (Few-Shot) Positive Beispiele für korrekte Bezeichner werden vorangestellt (z. B. „EnrolledStudents“, „numberOfValidCreditCards“).

V3 (Regeln) Kurzregeln zu CamelCase, Verben für Methoden, Nominalphrasen für Klassen.

V4 (Erklärung) Kurze Definitionen der Kategorien A–F (linguistische Anti-Muster) werden vorangestellt; die Antwort soll knapp Hinweise für Programmierer liefern, keine automatische Code-Änderung.

3.4 Durchführung der Tests

Für jedes der 10–15 fehlerhaften Beispiele und für jedes der 5–7 korrekten Beispiele wurde jede Prompt-Version an alle drei Modelle über HAWKI gesendet. Jede Modellantwort wurde protokolliert und anonymisiert abgespeichert. Der Ablauf:

1. Beispiel auswählen und Kategorie zuordnen.
2. Prompt-Version wählen.
3. Anfrage an Modell via HAWKI senden.
4. Ausgabe speichern.
5. Bewertung durchführen.

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?

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?

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

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: https://www.veneraarnaoudova.ca/wp-content/uploads/2014/10/2014-EMSE-Arnaodova-et-al-Perception-LAs.pdf, abgerufen am 14.10.2025.

[2] Oracle Corporation: *Naming Conventions.* Verfügbar unter: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html, abgerufen am 14.10.2025.

[3] Lyu, M. R.; Rajan, B.; Roychoudhury, A.; Tan, S. H.; Thummalapenta, P.: *Automatic Programming: Large Language Models and Beyond.* Verfügbar unter: https://dl.acm.org/doi/pdf/10.1145/3708519, abgerufen am 15.11.2025.

[4] Butler, S.; Wermelinger, M.; Yu, Y.; Sharp, H. (2010): *Exploring the Influence of Identifier Names on Code Quality: An Empirical Study.* In: Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR), 15–18 March 2010, Madrid, Spain. Verfügbar unter: https://www.researchgate.net/publication/42799923_Exploring_the_Influence_of_Identifier_Names_on_Code_Quality_An_Empirical_Study, abgerufen am 26.11.2025.

[5] Ghosh Paul, D.; Zhu, H.; Bayley, I. (2025): *Investigating the Smells of LLM Generated Code.* School of Engineering, Computing and Mathematics, Oxford Brookes University, October 2025. Verfügbar unter: https://www.researchgate.net/publication/396223444_Investigating_The_Smells_of_LLM_Generated_Code, abgerufen am 26.11.2025.

[6] Santa Molison, A.; Moraes, M.; Melo, G.; Santos, F.; Assunção, W. K. G. (2025): *Is LLM-Generated Code More Maintainable & Reliable than Human-Written Code?* Toronto Metropolitan University; Colorado State University; North Carolina State University, July 2025. Verfügbar unter: https://www.researchgate.net/publication/393853113_Is_LLM-Generated_Code_More_Maintainable_Reliable_than_Human-Written_Code, abgerufen am 26.11.2025.

[7] Akram, W.; Jiang, Y.; Zhang, Y.; Khan, H. A.; Liu, H. (2025): *LLM-Based Method Name Suggestion with Automatically Generated Context-Rich Prompts.* Beijing Institute of Technology; Peking University. Verfügbar unter: https://www.researchgate.net/publication/392855381_LLM-Based_Method_

Name_Suggestion_with_Automatically_Generated_Context-Rich_Prompts, abgerufen am 27.11.2025.

[8] Andrade, R.; Torres, J.; Ortiz-Garcés, I. (2025): *Enhancing Security in Software Design Patterns and Antipatterns: A Framework for LLM-Based Detection.* In: Electronics, 14(3) (2025), Artikel Nr. 586. DOI: https://doi.org/10.3390/electronics14030586. Submission received: 16.11.2024; revised: 21.01.2025; accepted: 28.01.2025; published: 01.02.2025.