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.

Diese Arbeit untersucht, wie zuverlässig Sprachmodelle Java-Bezeichner beurteilen können und in welchen Fällen menschliches Eingreifen weiterhin notwendig bleibt. Bewertet werden Verständlichkeit, Bedeutung, Einhaltung von Namenskonventionen und das Erkennen linguistischer Anti-Muster [4]. Unter linguistischen Anti-Mustern versteht man systematische sprachliche Probleme in Bezeichnern, die Klarheit und Lesbarkeit beeinträchtigen.

Das Ziel dieser Arbeit besteht darin, zu überprüfen, ob Sprachmodelle hilfreiche Hinweise zur Verbesserung von Bezeichnern liefern können. Dazu werden mehrere Prompt-Varianten entwickelt, getestet und schrittweise verbessert. Die Untersuchung erfolgt anhand von Java-Beispielen mit fehlerhaften sowie korrekten Bezeichnern, um sowohl korrekte Erkennungen als auch mögliche Fehlalarme („False Positives“) zu beurteilen.

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).** Beschreibung der Methode zur Untersuchung von Sprachmodellen, Erstellung von Java-Beispielen mit schlechten Bezeichnern und Bewertung der Ergebnisse. Eine automatische Pipeline wird nur theoretisch erwähnt.
4. **Kapitel 4 (Evaluation).** Auswertung der Ergebnisse.
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 [3]. Sie dürfen nicht mit einer Ziffer beginnen und keine reservierten Wörter wiederholen. In der Praxis orientieren sich gute Namen an ihrer Funktion im Programm und an allgemein anerkannten Konventionen.

2.2 Namenskonventionen in Java

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

- Klassen und Schnittstellen. Bezeichner für Klassen und Schnittstellen werden als Nominalphrasen formuliert. Jeder Wortanfang steht groß, z.B. `Student`, `EnrolledStudents`. 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`.

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.

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

2.3 Linguistische Anti-Muster

Linguistische Anti-Muster (LA) 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* [4] 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, wie zuverlässig Sprachmodelle Java-Bezeichner bewer-

ten 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:

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

Kategorie D2. Prädikatname, Typ nicht Boolean:

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

Kategorie E1. Pluralname bei singular Typ:

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

Kategorie F1. Name und Typ im Gegensatz:

```java
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?

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] European Conference on Software Maintenance and Reengineering (2013): *A New Family of Software Anti-Patterns: Linguistic Anti-Patterns.* Konferenzbeitrag CSMR 2013. Verfügbar unter: https://assets.ptidej.net/Publications/Documents/CSMR13d.doc.pdf, abgerufen am 15.11.2025.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.