



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

1	Einleitung	3
2	Theoretische Hintergrund	5
2.1	Java-Bezeichner	5
2.2	Namenskonventionen in Java	5
2.3	Linguistische Anti-Muster	5
2.4	Große Sprachmodelle (LLMs)	6
3	Methodik und Durchführung	7
3.1	Auswahl der Sprachmodelle	7
3.2	Erstellung der Test-Beispiele	7
3.2.1	Kategorien nach Arnaoudova	7
3.2.2	Beispiel-Code	7
3.2.3	Korrekte Beispiele für False-Positive-Tests	8
3.3	Hinweis-Strategien und Iteration	8
3.4	Iteratives Vorgehen	9
3.5	Durchführung der Tests	10
3.6	Bewertungskriterien	10
	Literatur	11

Kapitel 1

1 Einleitung

Sprachmodelle werden in der Softwareentwicklung immer häufiger eingesetzt, um Entwickler bei Analyse- und Review-Aufgaben zu unterstützen. Ein wichtiger Anwendungsbe-
reich 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 Hinweis-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 Hinweis-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 Rückgabety `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

3.1 Auswahl der Sprachmodelle

Für die Untersuchung wurden drei verschiedene Modelle über HAWKI¹ 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

Basierend auf Arnaoudova et al. [1] wurden zwei Arten von Beispielen erstellt:

- **10–15 Beispiele mit linguistischen Anti-Mustern** (alle Kategorien A–F),
- **5–7 korrekt benannte Beispiele** ohne Anti-Muster, um unbegründete Fehlalarme zu erkennen.

Jede Kategorie wurde kurz erklärt und durch ein prägnantes Beispiel illustriert.

3.2.2 Beispiel-Code

Kategorie A1 – „*tut mehr als der Name sagt*“. Methode mit `get` macht mehr als zurückgeben:

```
1 public class UserManager {  
2     private User currentUser;  
3  
4     public User getUser() {  
5         validateSession();    // Nebenwirkung  
6         return currentUser;  
7     }  
8 }
```

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

```
1 public class Calculator {  
2     public void getResult() {
```

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

```

3         System.out.println("42");
4     }
5 }

```

Kategorie C1. Widerspruch zwischen Name und Rückgabewert:

```

1 public class Config {
2     // Problem: disable() gibt ControlEnableState zurück
3     public ControlEnableState disable() {
4         return new ControlEnableState(true);
5     }
6 }

```

Kategorie D2. Prädikatname, Typ nicht Boolean:

```

1 public class Payment {
2     private int isActive; // Typ passt nicht zum Prädikat
3 }

```

Kategorie E1. Pluralname bei singular Typ:

```

1 public class Order {
2     private Product products; // Name suggeriert Mehrzahl
3 }

```

Kategorie F1. Name und Typ im Gegensatz:

```

1 public class Mode {
2     private boolean disabledState = true;
3 }

```

3.2.3 Korrekte Beispiele für False-Positive-Tests

```

1 public class StudentRecord {
2     private int enrolledStudents;
3 }

```

Diese Beispiele enthalten keine Anti-Muster. Ein korrektes Modell darf hier *keine* Warnung ausgeben [8].

3.3 Hinweis-Strategien und Iteration

Die Hinweise wurden iterativ nach Versionen überarbeitet: Version 1 (V1) → Version 2 (V2) → Version 3 (V3) → Version 4 (V4). Das Ziel jeder Iteration war die Qualitätssteigerung der Hinweise (automatischer Überblick) und die Reduzierung der Anzahl von

Fehlalarmen. V4 enthält auch eine kurze Definition linguistischer Anti-Muster, damit Modelle diese Kategorien zielorientiert erkennen können.

V1 (Zero-Shot)

Analysieren Sie die Bezeichner im folgenden Java-Code und identifizieren Sie mögliche Benennungs- oder Struktur-Anti-Muster.

V2 (Few-Shot)

Beispiele für gute Bezeichner:

- Klassen: CustomerAccount, PaymentProcessor
- Methoden: calculateTotal(), validateInput()
- Attribute: userName, itemCount

Analysieren Sie nun folgenden Code:

[CODE]

V3 (Kontextreicher Hinweis)

Analysieren Sie den Code unter Berücksichtigung von:

- 1) Java-Konventionen (Oracle)
- 2) Konsistenz zwischen Name, Typ und Verhalten
- 3) Verständlichkeit und Bedeutung

Geben Sie Hinweise für Entwickler.

V4 (Erklärung der linguistischen Anti-Muster)

Da unklar ist, ob Modelle linguistische Anti-Muster kennen [1], wird in dieser Version eine kurze Erklärung ergänzt.

Linguistische Anti-Muster:

- A: Methode tut mehr als Name sagt
- B: Name verspricht mehr als implementiert
- C: Name und Verhalten sind gegensätzlich
- D-F: Inkonsistenzen bei Attributnamen

Analysieren Sie nun den Code:

[CODE]

3.4 Iteratives Vorgehen

Der Hinweis wurde iterativ verbessert:

1. erste Tests mit Version 1,
2. Analyse der Fehlbewertungen,
3. Erweiterung um Beispiele (Version 2),
4. Hinzufügen von Kontextregeln (Version 3),
5. Ergänzung der Definitionen von Anti-Mustern (Version 4).

Dieses Vorgehen folgt aktuellen Forschungsergebnissen, die die Bedeutung von Hinweis-Gestaltung und Iteration bei der Codeanalyse hervorheben [7, 3].

3.5 Durchführung der Tests

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

3.6 Bewertungskriterien

Jedes Kriterium wird mit 1-5 Punkten bewertet.

1. **Anti-Muster-Erkennung.** Wurde das Problem erkannt?
2. **Korrekturqualität.** Ist der vorgeschlagene neue Name besser? Löst er das Anti-Muster?
3. **Konventions-Konformität.** Folgt der neue Code Java-Standards?
4. **Semantische Klarheit.** Sind die neuen Namen verständlich?
5. **Konsistenz.** Passt alles zusammen?

Zusätzlich wurden Fehlalarme („False Positives“) gesondert untersucht, da aktuelle Forschung zeigt, dass sie bei LLM-basierten Analysen häufig auftreten [8, 6].

Das Bewusstsein für linguistische Anti-Muster 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-Arnaoudova-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.