



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Kapitel 02

REFACTORING — VOM
MONOLITHEN ZU MODULEN

Wo wir jetzt stehen

Herzlichen Glückwunsch! Sie haben erreicht:

- ✓ **Kapitel 02 (Code-Qualität in der Praxis):** PEP8-Konformität mit Ruff
- ✓ **Kapitel 02 (Feature-Branch-Entwicklung):** Feature Branches & Pull Requests
- ✓ **Kapitel 02 (Automatisierung und CI/CD):** Automatisierte CI/CD-Checks

Ihre CI ist grün! 🎉

Aber... Ihre gesamte App lebt in einer einzigen 390-Zeilen `main.py` Datei.

Das Problem: Echte Schmerzpunkte

Szenario 1: Die Bug-Jagd

Sie: "Da ist ein Bug in der Schnittpunkt-Berechnung."
Sie: *scrollen durch 390 Zeilen*
Sie: *passieren generate_road_profile(), create_dash_app()...*
Sie: "Gefunden bei Zeile 267! Moment, wovon hängt das ab?"

Szenario 2: Der Merge-Konflikt

Student A: Bearbeitet Zeilen 50-120 (Straßengenerierung)
Student B: Bearbeitet Zeilen 200-350 (UI-Styling)
Ergebnis: MERGE-KONFLIKT (trotz unabhängiger Änderungen!)

Das Problem: Test-Albtraum

Szenario 3: Testen (Vorschau auf Kapitel 03 (Grundlagen des Testens))

```
# Sie wollen find_intersection() testen
from main import find_intersection

# Was tatsächlich passiert:
# - Importiert das gesamte Dash-Framework
# - Startet Webserver
# - Dauert 5 Sekunden pro Test
# - 20 Funktionen zu testen = 100 Sekunden!
```

Ergebnis: Sie geben das Testen auf. 😞

Heutige Lernziele

Am Ende dieser Vorlesung wirst Sie:

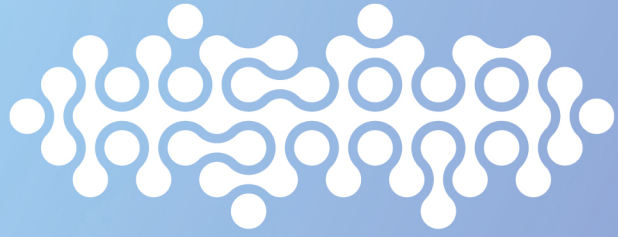
1.  **Verstehen** warum monolithischer Code unwartbar wird
2.  **Identifizieren** natürliche Grenzen in Ihrem Code
3.  **Refactoren** einen Monolithen in fokussierte Module
4.  **Organisieren** Imports um zirkuläre Abhängigkeiten zu vermeiden
5.  **Erleben** die Vorteile von modularem Code
6.  **Vorbereiten** Ihre Codebasis fürs Testen (Kapitel 03 (Grundlagen des Testens))

Was Sie heute NICHT lernst

Nicht behandelt (noch nicht):

- ✗ Design Patterns mit Namen (ergeben sich natürlich aus der Praxis)
- ✗ Fortgeschrittene Architekturmuster (kommen später)
- ✗ Testen (das ist Kapitel 03 (Grundlagen des Testens)!)

Heutiger Fokus: Praktisches Refactoring mit echtem Code.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Workflow & Plan

1

Feature-Development-Workflow anwenden

Erinnern Sie sich an den Workflow aus Kapitel 02 (Feature-Entwicklung)?

1. Erstelle einen Feature-Branch
2. Implementiere Ihre Änderungen
3. Pushe zu GitHub
4. Öffne einen Pull Request
5. Automatisierte CI-Checks laufen
6. Code Review
7. Merge bei Genehmigung
8. Lösche den Branch

Heutiges Refactoring IST ein Feature! Wir nutzen diesen Workflow.

Schritt 1: Feature-Branch erstellen

```
# Stelle sicher, dass Sie auf main bist und aktuell
$ git checkout main
$ git pull origin main

# Erstelle einen neuen Feature-Branch
$ git checkout -b feature/refactor-to-modules
Switched to a new branch 'feature/refactor-to-modules'

# Überprüfe, dass Sie auf dem neuen Branch bist
$ git branch
  main
* feature/refactor-to-modules
```

Warum? Main bleibt stabil während wir sicher experimentieren.

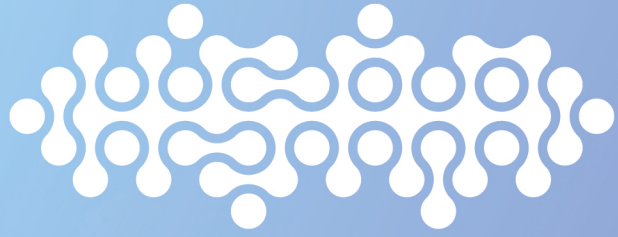
Der Plan: Iteratives Refactoring

Kleine, sichere Schritte mit Commits nach jedem:

```
Schritt 1: Extrahiere geometry.py → Commit → Test  
Schritt 2: Extrahiere road.py → Commit → Test  
Schritt 3: Extrahiere visualization.py → Commit → Test  
Schritt 4: Vereinfache main.py → Commit → Test  
Final: Pushe Branch → Öffne PR → CI validiert → Merge
```

Schlüsselprinzip: Nach jedem Schritt funktioniert die Anwendung noch perfekt.

Warum kleine Schritte? Leicht zu erkennen was kaputtging, leicht rückgängig zu machen.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Den Monolithen verstehen

2

Was ist ein Monolith?

Code, wo alles an einem Ort lebt, eng miteinander gekoppelt.

Ihr aktuelles `main.py` (390 Zeilen):

- Straßenprofil-Generierung (Zeilen 20-60)
- Geometrie-Berechnungen (Zeilen 60-180)
- Dash-Anwendungs-UI (Zeilen 190-390)
- Anwendungs-Einstiegspunkt

Alles ist durcheinander: Mathe, UI, Datengenerierung, App-Logik.

Warum Monolithen zu Problemen werden

1. Schwer zu navigieren

- Spezifische Funktionalität zu finden erfordert durch unabhängigen Code zu scrollen
- Zeile 111? War das Geometrie oder Straßengenerierung?

2. Schwer zu testen

- Muss alles importieren um eine Funktion zu testen
- Dash startet, langsame Imports, Seiteneffekte

3. Schwer zusammenzuarbeiten

- Zwei Entwickler bearbeiten dieselbe Datei = Merge-Konflikte
- Auch wenn die Änderungen völlig unabhängig sind!

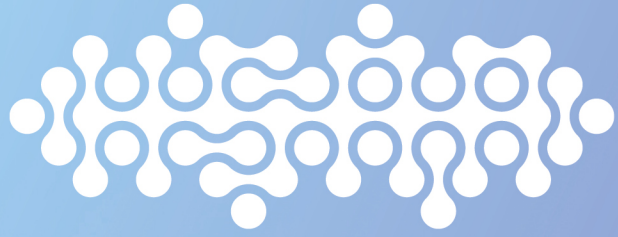
Warum Monolithen zu Problemen werden (2)

4. Hohe Kopplung

```
def find_intersection(...):  
    # Verwendet versehentlich eine globale Variable aus dem UI-Bereich  
    camera_x = CAMERA_POSITION # Wo ist das definiert??  
    # Jetzt hängt Geometrie von UI ab - schwer zu testen!
```

5. Verletzt "Eine Datei, Ein Zweck"

- Was macht `main.py` ?
- Aktuelle Antwort: "Alles!"
- Bessere Antwort: "Startet die Anwendung. Das war's."



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Natürliche Grenzen

3

Der "3-Wörter-Beschreibung"-Test

Wenn Sie ein Modul in 3 Wörtern oder weniger beschreiben kannst, ist es fokussiert genug.

✓ Gute Beispiele:

- `geometry.py` → "Strahlenschnitt-Mathematik"
- `road.py` → "Straßenprofil-Generierung"
- `visualization.py` → "Dash UI-Komponenten"
- `main.py` → "Anwendungs-Einstiegspunkt"

Der "3-Wörter-Test" (Fortsetzung)

✗ Schlechte Beispiele:

- `utilities.py` → "Verschiedene nützliche Sachen" (zu vage!)
- `helpers.py` → "Hilfsfunktionen" (welche Art??)
- `functions.py` → "Verschiedene Funktionen" (bedeutungslos!)

Schlüsselprinzip: Spezifische, fokussierte Benennung zeigt klaren Zweck.

Ihren aktuellen Code analysieren

Kategorie 1: Reine Mathe-Funktionen

- `calculate_ray_line()` - Hängt nur von numpy ab
- `find_intersection()` - Hängt nur von numpy ab
- **Charakteristik:** Gleiche Eingaben = gleiche Ausgaben (rein)
- **Ziel:** `geometry.py`

Kategorie 2: Datengenerierung

- `generate_road_profile()` - Hängt nur von numpy ab
- **Charakteristik:** Generiert Daten, zeigt sie aber nicht an
- **Ziel:** `road.py`

Ihren aktuellen Code analysieren (2)

Kategorie 3: UI und Anwendungslogik

- `create_dash_app()` - Hängt von Dash, Geometrie, Road ab
- `update_graph()` Callback - Orchestriert alles
- **Charakteristik:** Erstellt UI, behandelt Benutzereingaben
- **Ziel:** `visualization.py`

Kategorie 4: Einstiegspunkt

- `main()` - Startet nur die Anwendung
- **Charakteristik:** Minimale Logik
- **Ziel:** `main.py` (einfach halten!)

Die Abhängigkeits-Hierarchie

Natürlicher Abhängigkeitsfluss:

```
main.py
  ↓ abhängig von
visualization.py
  ↓ abhängig von
geometry.py, road.py
  ↓ abhängig von
numpy (externe Bibliothek)
```

Schlüssel-Einsicht: Abhängigkeiten fließen **NUR IN EINE RICHTUNG**.

Warum? Verhindert zirkuläre Abhängigkeiten!

✓ `visualization.py` kann aus `geometry.py` importieren

✗ `geometry.py` sollte NICHT aus `visualization.py` importieren

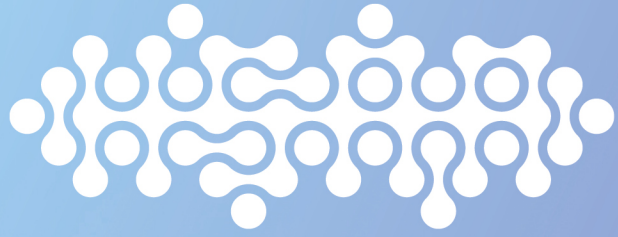
Zielstruktur

Nach dem Refactoring:

```
road-profile-viewer/  
├─ main.py           # Nur Einstiegspunkt (~15 Zeilen)  
├─ geometry.py       # Reine Mathe-Funktionen (~80 Zeilen)  
├─ road.py           # Straßengenerierung (~60 Zeilen)  
├─ visualization.py  # Dash-App und UI (~200 Zeilen)  
└─ (andere Dateien bleiben gleich)
```

Von 1 Datei (390 Zeilen) → 4 fokussierte Module (465 Zeilen gesamt)

Zusätzliche Zeilen sind Dokumentation - das ist gut! 



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Praktisches Refactoring

4

Schritt 1: Modul-Dateien erstellen

```
# In Ihrem road-profile-viewer-Verzeichnis
$ touch geometry.py
$ touch road.py
$ touch visualization.py

# Überprüfe die Struktur
$ ls *.py
geometry.py  main.py  road.py  visualization.py
```

Jetzt haben wir unsere Modul-Dateien bereit!

Schritt 2: Geometrie extrahieren

Erstellen Sie `geometry.py` mit:

- `calculate_ray_line()` Funktion
- `find_intersection()` Funktion
- Umfassende Docstrings
- Type Hints für IDE-Unterstützung

Aktualisieren Sie `main.py` :

```
from geometry import calculate_ray_line, find_intersection
```

Löschen Sie die Geometrie-Funktionsdefinitionen aus `main.py` .

Schritt 2: Testen & Committen

Testen Sie, dass es funktioniert:

```
$ uv run road-profile-viewer  
# Anwendung sollte normal starten  
# Überprüfe, dass Strahlenschnitt noch funktioniert
```

✓ **Checkpoint:** Anwendung funktioniert!

Committe Ihren Fortschritt:

```
$ git add geometry.py main.py  
$ git commit -m "Extract geometry functions to geometry.py"
```

- Move `calculate_ray_line()` and `find_intersection()`
- Add comprehensive docstrings and type hints
- Update `main.py` imports
- Application still works identically"

Schritt 3: Road extrahieren

Erstellen Sie `road.py` **mit:**

- `generate_road_profile()` Funktion
- Docstrings, die die Klothoiden-Approximation erklären

Aktualisieren Sie `main.py` :

```
from road import generate_road_profile
```

Löschen Sie die Straßengenerierungs-Funktion aus `main.py` .

Schritt 3: Testen & Committen

Testen Sie erneut:

```
$ uv run road-profile-viewer  
# Überprüfe, dass Straßengenerierung noch funktioniert
```

✓ **Checkpoint:** Funktioniert noch!

Committe diesen Schritt:

```
$ git add road.py main.py  
$ git commit -m "Extract road generation to road.py"
```

- Move `generate_road_profile()`
- Add docstrings
- Update imports
- Application still works identically"

Schritt 4: Visualization extrahieren

Erstellen Sie `visualization.py` mit:

- `create_dash_app()` Funktion
- Allem UI-Layout-Code
- Allen Callbacks
- Imports aus `geometry` und `road` Modulen

Vereinfachen Sie `main.py` :

```
from visualization import create_dash_app

def main():
    app = create_dash_app()
    print("Starting Road Profile Viewer...")
    app.run(debug=True)
```

Schritt 4: Testen & Committen

Testen Sie das komplette Refactoring:

```
$ uv run road-profile-viewer  
# Überprüfe dass ALLES noch funktioniert:  
# - Straße wird korrekt gerendert  
# - Strahlwinkel passt sich an  
# - Schnittpunkt aktualisiert sich  
# - Distanz wird angezeigt
```

✓ **Checkpoint:** Alles funktioniert! 🎉

Schritt 4: Finaler Commit

```
$ git add visualization.py main.py  
$ git commit -m "Extract UI layer to visualization.py"
```

- Move `create_dash_app()` and all UI code
- Simplify `main.py` to just entry point (~20 lines)
- `visualization.py` imports from `geometry` and `road`
- Application works identically to before

Completes modular refactoring:
main → visualization → geometry/road"

Jetzt haben wir eine saubere Commit-Historie, die die Refactoring-Progression zeigt!

Das Refactoring überprüfen

Überprüfen Sie Ihre Modul-Struktur:

```
$ ls *.py  
geometry.py  main.py  road.py  visualization.py
```

Überprüfen Sie Zeilenzahlen:

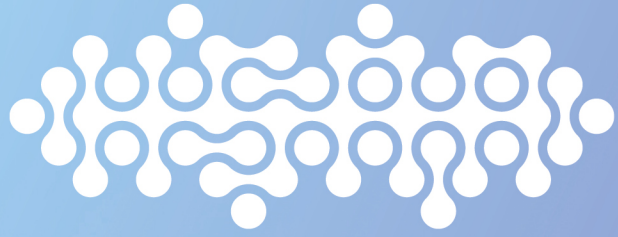
```
$ wc -l *.py  
175 geometry.py      # Reine Geometrie-Berechnungen  
67  road.py          # Straßengenerierung  
203 visualization.py # Aller UI-Code  
20  main.py          # Nur der Einstiegspunkt  
---  
465 total
```

main.py ging von 390 Zeilen auf 20 Zeilen! 

Abhängigkeitsfluss überprüfen

```
# main.py importiert:  
from visualization import create_dash_app  
  
# visualization.py importiert:  
from geometry import calculate_ray_line, find_intersection  
from road import generate_road_profile  
  
# geometry.py importiert:  
import numpy as np # Nur externe Abhängigkeit  
  
# road.py importiert:  
import numpy as np # Nur externe Abhängigkeit
```

✓ **Einseitige Abhängigkeiten - keine Zyklen!**



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Gewonnene Vorteile

5

Einfache Navigation

Vorher:

```
# "Wo ist die Schnittpunkt-Berechnung?"  
# *Scrollt durch 390 Zeilen*  
# *Verirrt sich in unabhängigem Code*
```

Nachher:

```
# "Wo ist die Schnittpunkt-Berechnung?"  
# Öffnet geometry.py → find_intersection() ist direkt da!
```

Gewinn: Code in Sekunden statt Minuten finden.

Fokussierte Änderungen

Vorher:

```
# Änderung des Straßengenerierungs-Algorithmus  
# Datei: main.py (390 Zeilen)  
# Risiko: Könnte versehentlich UI-Code kaputt machen
```

Nachher:

```
# Änderung des Straßengenerierungs-Algorithmus  
# Datei: road.py (67 Zeilen)  
# Sicherheit: Kann UI-Code nicht versehentlich anfassen!
```

Gewinn: Änderungen sind sicherer und fokussierter.

Bessere Zusammenarbeit

Vorher:

Student A: Bearbeitet main.py Zeilen 50-120 (Straßengenerierung)
Student B: Bearbeitet main.py Zeilen 200-350 (UI-Styling)
Ergebnis: MERGE-KONFLIKT! 😡
Zeitverschwendung: 30 Minuten Konflikte auflösen

Nachher:

Student A: Bearbeitet road.py
Student B: Bearbeitet visualization.py
Ergebnis: Keine Konflikte! 😊
Zeitverschwendung: 0 Minuten

Gewinn: Parallele Entwicklung ohne Konflikte!

Testbarkeit

Vorher:

```
# Um find_intersection() zu testen
from main import find_intersection
# Importiert ALLES (Dash, UI, etc.)
# Langsam, schwer, Seiteneffekte
```

Nachher:

```
# Um find_intersection() zu testen
from geometry import find_intersection
# Importiert nur Geometrie-Modul
# Schnell, leichtgewichtig, keine Seiteneffekte
```

Gewinn: Schnelle, fokussierte Tests (Kapitel 03 (Grundlagen des Testens)!)

Wiederverwendbarkeit

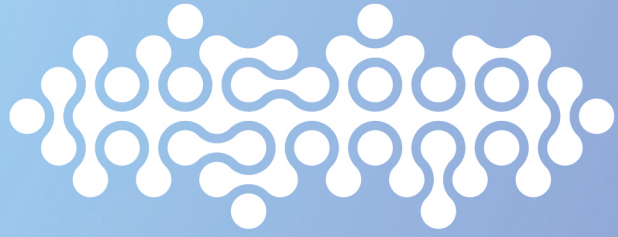
Vorher:

```
# Willst Sie find_intersection() in einem anderen Projekt nutzen?  
# Copy-Paste aus main.py  
# → Bringt unnötigen Dash-Code mit 😞
```

Nachher:

```
# Willst Sie find_intersection() in einem anderen Projekt nutzen?  
# Kopiere einfach geometry.py  
# → Sauberes, eigenständiges Modul! 😊
```

Gewinn: Reine Funktionen sind überall wiederverwendbar.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Häufige Fallstricke

6

Fallstrick #1: Zirkuläre Imports

Der Fehler:

```
ImportError: cannot import name 'generate_road_profile'  
from partially initialized module 'road'  
(most likely due to a circular import)
```

Die Ursache:

```
# visualization.py  
from road import generate_road_profile  
  
# road.py (SCHLECHT!)  
from visualization import create_dash_app  
# Warum sollte road.py UI-Code brauchen??
```

Fallstrick #1: Lösung

Regel: Module niedrigerer Ebene (geometry, road) sollten **NIEMALS** aus Modulen höherer Ebene (visualization, main) importieren.

Abhängigkeiten fließen in eine Richtung:

```
main → visualization → geometry/road
```

Nicht:

```
geometry ← visualization ← main (✗ FALSCH!)
```

Fallstrick #2: Unklare Grenzen

Die Verwirrung:

```
# Wo gehören camera_x und camera_y hin?  
# - geometry.py? (werden in Berechnungen verwendet)  
# - visualization.py? (ist ein UI-Parameter)  
# - Eine neue config.py Datei?
```

Die Lösung:

- **Wenn hardcodiert:** Behalte als Parameter-Default in Funktionssignatur
- **Wenn vom Benutzer konfigurierbar:** Behalte in visualization.py (UI entscheidet)
- **Erstelle "config.py" nicht vorzeitig!** Sie werden wissen, wann es gebraucht wird.

Fallstrick #3: Über-Refactoring

Die Versuchung:

Sollte ich erstellen:

- geometry_helpers.py?
- geometry_utils.py?
- ray_calculations.py?
- intersection_math.py?
- angle_converter.py?

Die Lösung: YAGNI-Prinzip

- **"You Ain't Gonna Need It"** (Sie werden es nicht brauchen)
- Beginnen Sie mit 3-4 fokussierten Modulen
- Teile nur, wenn Modul zu groß wird (>300 Zeilen)
- **Vermeide "utility/helper"-Module** - sie werden zu Müllkippen!

Fallstrick #4: Imports vergessen

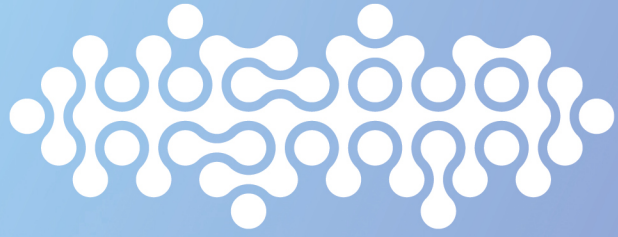
Der Fehler:

```
# In main.py
def main():
    x, y = generate_road_profile()
# NameError: name 'generate_road_profile' is not defined
```

Die Lösung:

```
# Füge den Import am Anfang hinzu
from road import generate_road_profile
```

Pro-Tipp: Nutzen Sie IDE Auto-Import (Ctrl+. in VS Code).



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

CI & PR-Workflow

7

CI-Checks ausführen

Ihr Refactoring sollte alle CI-Checks bestehen:

```
# Führe Ruff aus (Style-Check)
```

```
$ uv run ruff check .
```

```
All checks passed! ✓
```

```
# Führe Ruff Format-Check aus
```

```
$ uv run ruff format --check .
```

```
All files formatted correctly! ✓
```

```
# Führe Pyright aus (Type-Check)
```

```
$ uv run pyright
```

```
0 errors, 0 warnings ✓
```

Wenn Sie Fehler bekommst: Behebe sie vor dem Pushen!

Ihren Feature-Branch pushen

Überprüfen Sie Ihre Commits:

```
$ git log --oneline  
7h8i9j0 Extract UI layer to visualization.py  
d4e5f6g Extract road generation to road.py  
a1b2c3d Extract geometry functions to geometry.py
```

Saubere, logische Progression! ✨

Pushen Sie zu GitHub:

```
$ git push -u origin feature/refactor-to-modules
```

Pull Request erstellen

```
$ gh pr create \  
  --title "Refactor: Split monolithic main.py into focused modules" \  
  --body "Refactors monolithic main.py (390 lines) into 4 modules:"
```

- geometry.py: Pure math functions
- road.py: Road generation logic
- visualization.py: Dash UI layer
- main.py: Entry point only (20 lines)

Benefits: Easier navigation, testing, reduced merge conflicts

- ✓ All Ruff/Pyright checks pass
- ✓ Application works identically"

Oder nutze das GitHub Web-Interface!

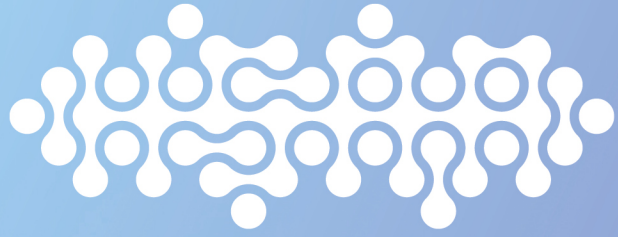
Auf CI-Checks warten

GitHub Actions wird automatisch laufen:

- ✓ Ruff check (style)
- ✓ Ruff format check
- ✓ Pyright (types)

Alle sollten bestehen! (Sie haben sie lokal ausgeführt.)

Sobald grün: Bereit zum Mergen! 🚀



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

KI als Werkzeug

8

KI-Coding-Assistenten nutzen

Wichtiger Hinweis:

Bevor Sie KI für Refactoring verwendest, MUSST Sie verstehen:

-  Was gute Modul-Grenzen ausmacht
-  Wie man zirkuläre Abhängigkeiten vermeidet
-  Der "3-Wörter-Beschreibung"-Test
-  Einseitiger Abhängigkeitsfluss

Warum? Ohne diese Prinzipien kannst Sie nicht:

- Effektive Prompts schreiben
- Vorschläge der KI bewerten
- Subtile Fehler erkennen

KI ist ein Werkzeug, kein Ersatz für Ihr Wissen!

Was KI kann und was nicht

KI kann helfen bei:

-  Mechanischer Code-Verschiebung (copy/paste zwischen Dateien)
-  Import-Anweisungen aktualisieren
-  Boilerplate schreiben (Docstrings, Type Hints)
-  Konsistenz wahren

KI kann nicht:

-  Entscheiden WO Code geteilt wird (erfordert Urteilsvermögen)
-  Architektur-Bedürfnisse Ihres Projekts verstehen
-  Abwägungen bewerten
-  Übereinstimmung mit Team-Konventionen sicherstellen

Beispiel-Prompt

Sie haben das Refactoring geplant (manuell!), jetzt hilft KI bei der Ausführung.

Sie werden die monolithische main.py (390 Zeilen) refactoren in:

- geometry.py - Reine Mathe-Funktionen
- road.py - Straßenprofil-Generierung
- visualization.py - Dash UI-Schicht
- main.py - Einstiegspunkt (~20 Zeilen)

Erstellen Sie Feature-Branch feature/refactor-to-modules

Machen Sie 4 inkrementelle Commits (einen pro Schritt).

Schritt 1: Geometrie-Modul extrahieren

- Kopieren Sie `calculate_ray_line()` und `find_intersection()`
- Fügen Sie Imports hinzu: `import numpy as np`
- Behalten Sie alle Docstrings und Type Hints
- Commit: "refactor: extract geometry to separate module"

[Weiter mit Schritten 2, 3, 4...]

Warum dieser Prompt funktioniert

Klare Struktur:

-  Explizite Dateinamen und Speicherorte
-  Schritt-für-Schritt-Aufschlüsselung
-  Spezifische Anforderungen
-  Commit-Message-Format

Enthält Kontext:

-  Erklärt WARUM (absolute Imports, etc.)
-  Was bewahrt werden soll (Docstrings)
-  Zeigt gewünschten Endzustand

Ermöglicht Verifikation:

-  Jeder Schritt unabhängig testbar

Klare Erfolgskriterien

Verantwortung nach Code-Generierung

Akzeptieren Sie niemals blind von KI generierten Code!

Immer:

1. ☒ Jede Zeile lesen - Verstehen was sich änderte
2. ☒ Nach jedem Schritt testen - `uv run road-profile-viewer`
3. ☒ Imports überprüfen - Auf zirkuläre Abhängigkeiten prüfen
4. ☒ CI-Checks ausführen - Sicherstellen Ruff, Pyright bestehen
5. ☒ Commits überprüfen - git diff kontrollieren
6. ☒ Struktur validieren - Entspricht es Ihrem Plan?

Warnzeichen beachten

Häufige KI-Fehler:

- ✗ Zirkuläre Imports (geometry importiert aus visualization)
- ✗ Fehlende oder falsche Import-Pfade
- ✗ Verlorene Funktionalität (gelöscht statt verschoben)
- ✗ Geänderte Logik (KI hat unnötig "verbessert")
- ✗ Inkonsistente Benennung oder Stil

Sie sind die Qualitätskontrolle!

Mensch-KI-Kollaborationsmodell

Mensch: Strategisches Denken

- |— Monolith-Probleme identifizieren
- |— Modul-Grenzen entscheiden
- |— Abhängigkeitshierarchie planen
- |— Erfolgskriterien definieren

↓

KI: Taktische Ausführung

- |— Code zwischen Dateien verschieben
- |— Boilerplate-Code schreiben
- |— Commit-Messages formatieren

↓

Mensch: Verifikation & Qualität

- |— Funktionalität testen
- |— Architektur überprüfen
- |— Nach subtilen Bugs suchen
- |— Gegen CI validieren

Sie sind der Architekt. KI ist die Baufirma.

Lernübungs-Empfehlung

Für Studenten:

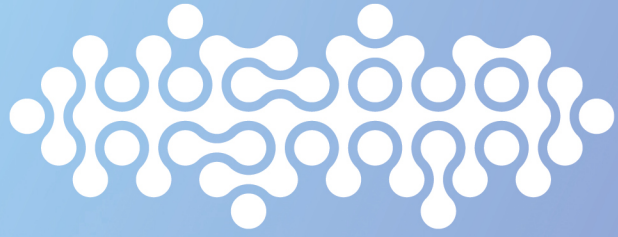
1. **Erstes Mal:** Manuell durchführen (heutige Übung)

- Herausforderungen tiefgreifend verstehen
- Intuition für Probleme aufbauen
- Lernen, zirkuläre Abhängigkeiten zu erkennen

2. **Zweites Mal:** KI-Unterstützung nutzen (zukünftige Projekte)

- Jetzt kannst Sie KI-Vorschläge bewerten
- Sie werden ihre Fehler erkennen
- Sie werden bessere Prompts schreiben

Warum diese Reihenfolge? Lerne zuerst Prinzipien, dann nutze Automatisierung.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Zusammenfassung

9

Vorher vs. Nachher

Vor dem Refactoring:

```
main.py (390 Zeilen)
└─ Alles durcheinander:
    Mathe, UI, Daten, Startup
```

- ✗ Schwer Code zu finden
- ✗ Schwer zu testen
- ✗ Merge-Konflikte
- ✗ Enge Kopplung

Nach dem Refactoring:

```
└─ geometry.py (175 Zeilen)
└─ road.py (67 Zeilen)
└─ visualization.py (203 Zeilen)
└─ main.py (20 Zeilen)
```

- ✓ Klare Modul-Grenzen
- ✓ Testbar
- ✓ Parallel editierbar
- ✓ Lose Kopplung

Vorteile

- ✓ **Einfache Navigation** - Klare Modul-Namen
- ✓ **Testbar** - Geometrie ohne Dash importieren
- ✓ **Kollaborativ** - Verschiedene Dateien gleichzeitig bearbeiten
- ✓ **Wiederverwendbar** - Reine Funktionen in eigenständigen Modulen
- ✓ **Wartbar** - Module mit einzelner Verantwortung

Gelernte Schlüsselprinzipien

1. **Separation of Concerns (Trennung der Zuständigkeiten)**

- Jedes Modul macht eine Sache gut

2. **Abhängigkeits-Richtung**

- Module niedrigerer Ebene importieren nicht aus höherer Ebene

3. **Der "3-Wörter-Beschreibung"-Test**

- Kannst Sie das Modul prägnant beschreiben?

4. **YAGNI ("You Ain't Gonna Need It")**

- Erstelle keine Abstraktionen bis sie gebraucht werden

Wichtige Erkenntnisse

1. **Monolithen wachsen organisch** - Anfangs in Ordnung, später Probleme
2. **Refactoring $\neq$ Neuschreiben** - Gleiche Funktionalität, bessere Struktur
3. **Module brauchen klare Grenzen** - Reine Funktionen vs. UI
4. **Abhängigkeiten fließen in eine Richtung** - Verhindert zirkuläre Imports
5. **Testbarkeit ergibt sich aus Modularität** - Kann nicht testen was man nicht importieren kann
6. **Der "3-Wörter-Test"** - Beschreibe Modul-Zweck prägnant
7. **YAGNI** - Beginne einfach, refactorere wenn nötig
8. **CI sollte bestehen** - Refactoring ändert nicht die Funktionalität

Weiterführende Literatur

Über Refactoring:

- Martin Fowler's "Refactoring: Improving the Design of Existing Code"
- Zitat: *"Jeder Dummkopf kann Code schreiben, den ein Computer versteht. Gute Programmierer schreiben Code, den Menschen verstehen."*

Über Modul-Design:

- Robert C. Martin's "Clean Code"
- SOLID-Prinzipien (Sie wenden Single Responsibility an!)

Über Python-Module:

- Real Python: Python Modules and Packages
- PEP 8: Package and Module Names

Übungszeit: Praktische Aufgabe!

Jetzt sind Sie an der Reihe, das Gelernte anzuwenden!

Was Sie tun wirst:

-  Eine echte monolithische Codebasis refactoren (390 Zeilen → 4 Module)
-  Die genauen Schritte aus der heutigen Vorlesung befolgen
-  Automatisiertes Feedback zu Ihrem Refactoring erhalten
-  Ihre Punktzahl steigen sehen bei jedem abgeschlossenen Schritt

Erlebe den vollständigen Workflow: Feature-Branch → Commits → Tests → CI/CD



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Fragen?

VIELEN DANK