Technische Hochschule Deggendorf
Faculty Applied Information Technology
Studiengang Bachelor of Cyber-Security

Projektimplementierung Reflexion zum Projekt "WeedDetector"

Vorgelegt von:

Christof Renner
Matrikelnummer: 22301943
Manuel Friedl
Matrikelnummer: 1236626

Prüfungsleitung:

Prof. Dr. Holger Jehle

Am: 11. Mai 2025

Contents

1 Controller-Implementierung

Die WeedDetector-Anwendung folgt dem Model-View-Controller (MVC) Entwurfsmuster, wobei der `WeedDetectorController` als zentraler Koordinator zwischen GUI und Erkennungsmodell fungiert.

Die Hauptaufgaben des Controllers umfassen:

- Ereignisbasierte Kommunikation zwischen GUI und Modell
- Verarbeitung von Benutzeraktionen (Bildauswahl, Erkennung, Robotersteuerung)
- Implementierung robuster Fehlerbehandlungsmechanismen

Die Kommunikation erfolgt über Callback-Funktionen, die eine lose Kopplung der Komponenten ermöglichen:

Listing 1.1: Callback-Registrierung

```
1 # GUI-Callbacks werden im Konstruktor registriert
2 self.gui.on_select_image = self.handle_select_image
3 self.gui.on_detect = self.handle_detect
4 self.gui.on_start_robot = self.handle_start_robot
5 self.gui.on_stop_robot = self.handle_stop_robot
```

2 Model & Modeltraining

Der WeedDetector nutzt ein YOLO-Modell für die Unkrauterkennung aufgrund seiner exzellenten Balance aus Geschwindigkeit und Genauigkeit. Die Entscheidung basiert auf einem umfassenden Vergleich verschiedener Objekterkennungsalgorithmen.

Die `WeedDetectorModel`-Klasse implementiert:

- Automatische Modellsuche mit Fallback-Mechanismus
- Konfigurierbare Erkennungsparameter (Konfidenz, IoU)
- Methoden für Training, Inferenz und Ergebnisvisualisierung

Der Trainingsprozess nutzt die Ultralytics-YOLO-API und einen umfangreichen Datensatz mit über 13.000 annotierten Bildern:

Listing 2.1: Training

```
1 def train(self, train_data_path, epochs=50):
2     """Train the YOLO model with provided training data."""
3     self.model.train(data=train_data_path, epochs=epochs)
```

Die Trainingsdaten wurden mit verschiedenen Augmentierungstechniken aufbereitet:

- Horizontale Spiegelung (50% Wahrscheinlichkeit)
- Zufällige Rotation und Bildausschnitte
- Belichtungsanpassungen

3 Security Scanning & Linting

3.1 Security Scanning mit Bandit

Für kontinuierliche Sicherheitsüberprüfungen wird Bandit eingesetzt, ein auf Python spezialisiertes Security-Scanning-Tool. Die Integration erfolgt über GitHub Actions:

```
name: Security Scan
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
jobs:
  bandit:
    name: Bandit Scan
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with: { python-version: '3.x' }
      - run: pip install bandit
      - run: bandit -r . --skip trojansource
```

Bandit analysiert den Code auf Sicherheitsprobleme wie Injektionsanfälligkeiten, Hartcodierte Credentials und Unsichere Funktionsaufrufe

3.2 Codequalität mit Pylint

Zur Sicherstellung konsistenter Codequalität wird Pylint als statisches Analysetool eingesetzt:

```
name: Pylint
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix: { python-version: ["3.13"] }
    steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-python@v5
      with: { python-version: ${{ matrix.python-version }} }
    - run: pip install pylint
    - run: pylint $(git ls-files '*.py')
```

Pylint prüft PEP 8 Konformität, Dokumentationsqualität, Komplexitätsmetriken und potenzielle Bugs.

4 Docker & Deployment

4.1 Containerisierung

Die Anwendung wird in Docker containerisiert, was eine konsistente Laufzeitumgebung sicherstellt. Die Dockerfile-Qualität wird mit Hadolint überprüft:

Listing 4.1: Docker Lint

```
name: Docker Build and Test
on:
  push:
    paths: [ 'Dockerfile' ]
  pull_request:
    paths: [ 'Dockerfile' ]
jobs:
  lint-dockerfile:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: hadolint/hadolint-action@v3.1.0
        with: { dockerfile: Dockerfile }
```

Der Container umfasst die Python-Laufzeitumgebung, YOLO- und OpenCV-Bibliotheken sowie die Anwendungskomponenten.

4.2 Automatisiertes Deployment

Das Deployment erfolgt automatisiert über GitHub Actions mit Integration in Docker Hub:

Listing 4.2: Build and Publish

```
name: Build and Publish Docker image
on:
  push:
    branches: [ main ]
    paths: [ 'Dockerfile' ]
jobs:
  build-and-push:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: docker/setup-buildx-action@v3
      - uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
      - uses: docker/build-push-action@v6
        with:
```

```
18            context: .
19            push: true
20            tags: crnnr/weeddetector:latest
```

4.3 Dependency-Management

Dependabot übernimmt die automatische Aktualisierung von Abhängigkeiten für Python-Pakete, Docker-Images und GitHub Actions.

Das Projekt unterstützt verschiedene Deploymentstrategien:

- Lokales Deployment für Entwicklung
- Container-basiertes Deployment für Produktion
- Versionierte Releases mit automatischen Updates