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

Projekt "WeedDetector" Arbeitsaufteilung Christof Renner

Vorgelegt von:

Christof Renner (22301943)
Manuel Friedl (1236626)

Prüfungsleitung:

Prof. Dr. Holger Jehle

Datum: 05. Juli 2025

Contents

1 Implementierung des Controllers

Die WeedDetector-Anwendung basiert auf dem Model-View-Controller Modell. Die Entscheidung für MVC fiel aufgrund etablierter Industry Standards und erhöht somit Wartbarkeit, Erweiterbarkeit und Testbarkeit der Software.

1.1 Zentrale Aufgaben des Controllers

Im Rahmen der Anwendung erfüllt der Controller folgende wesentliche Funktionen:

- **Ereignisgesteuerte Kommunikation:** Vermittelt zwischen GUI-Events und Modellaktionen, indem er Benutzereingaben weiterleitet und die Ergebnisse zurück zur GUI gibt.
- **Aktionsverarbeitung:** Reagiert auf spezifische Nutzeraktionen wie Bildauswahl, Start der Unkrauterkennung oder Steuerung des Roboters.
- **Robuste Fehlermechanismen:** Implementiert systematische Fehlerbehandlung zur Sicherstellung von Stabilität und Robustheit.

1.2 Lose Kopplung durch Callback-Funktionen

Um die Abhängigkeiten zwischen GUI und Controller minimal zu halten, nutzt der Controller sogenannte Callback-Funktionen, die bei bestimmten Ereignissen ausgelöst werden. Die Idee dahinter war die hohe Modularität des Vorgehens:

Listing 1.1: Callback-Registrierung im Controller

```
1 self.gui.on_select_image = self.handle_select_image
2 self.gui.on_detect = self.handle_detect
3 self.gui.on_start_robot = self.handle_start_robot
4 self.gui.on_stop_robot = self.handle_stop_robot
```

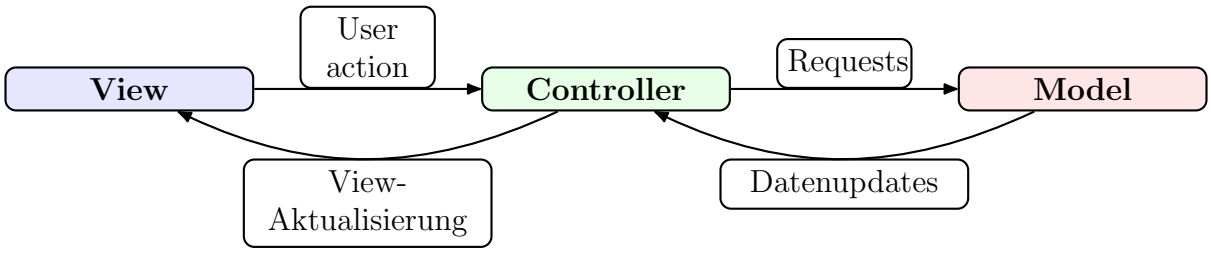

Figure 1.1: MVC-Architektur der WeedDetector-Anwendung

2 Modell, Dataset und Training

2.1 YOLO Modellarchitektur

Das verwendete YOLO-Modell wurde aufgrund seiner ausgewogenen Balance zwischen Geschwindigkeit und Genauigkeit gewählt. Es eignet sich hervorragend zur Echtzeiterkennung verschiedener Unkrautarten.

2.2 Datasetbeschreibung

Die Trainingsdaten umfassen 13.349 Bilder, exportiert über Roboflow im YOLO-Format (640×640 Pixel). Die Augmentierung wurde über die Online-Plattform "Roboflow" durchgeführt und die Bilder anschließend lokal trainiert.

2.3 Vorverarbeitungsschritte

Folgende Vorverarbeitungsschritte wurden durchgeführt:

- Auto-Orientierung zur Normalisierung der Bildausrichtung
- Größenanpassung auf 640×640 Pixel (Stretch)
- Automatische Kontrastanpassung mittels adaptiver Equalisierung

2.4 Bildaugmentierung

Zur Verbesserung der Generalisierung wurden umfangreiche Augmentierungen vorgenommen, sodass aus jedem Originalbild drei Trainingsvarianten entstanden:

- Horizontale Spiegelung mit 50% Wahrscheinlichkeit
- 90° Rotation (zufällig, Uhrzeiger- und Gegenuhrzeigersinn)
- Zufällige Rotation zwischen -15° und +15°
- Zufällige Ausschnitte (Crop) zwischen 0% und 20%
- Zufällige Anpassung von Sättigung (±20%) und Belichtung (±10%)

3 Security und Linting

3.1 Statische Codeanalyse

Wir setzen im Bereich Security auf statische Codeanalyse, dabei verwenden wir zur Analyse zum einen "Bandit" und "snyk". Die Entscheidung wurde dabei auf Basis von bisherigen Erfahrungen und aufgrund der weiten Verbreitung der Tools, gerade im Pythonumfeld, getätigt.

Listing 3.1: Bandit via GitHub Actions

```
1 - uses: actions/setup-python@v4
2   with: {python-version: '3.x'}
3 - run: pip install bandit
4 - run: bandit -r . --skip trojansource
```

Ergebnisse werden automatisiert bereitgestellt und im Falle von Auffälligkeiten als Issues behandelt.

3.2 Python Linting

Pylint analysiert kontinuierlich den Python-Code und prüft dabei auf Codequalität, Wartbarkeit, Fehleranfälligkeit und Einhaltung der PEP8-Konventionen. Auch wenn Linter nicht direkt securityrelevant sind, stellen wir so sicher, dass sauberer, strukturierter Code produziert wird.

Listing 3.2: Pylint Integration

```
1 - uses: actions/setup-python@v5
2   with: {python-version: '3.13'}
3 - run: pip install pylint
4 - run: pylint $(git ls-files '*.py')
```

3.3 Dockerfile Linting

Neben Python haben wir auch die Dockerfile mit Hadolint geprüft, um Best-Practices einzuhalten und potenzielle Schwachstellen auszuschließen:

Listing 3.3: Hadolint Action für Dockerfile

```
1 - uses: hadolint/hadolint-action@v3.1.0
2   with: {dockerfile: Dockerfile}
```

3.4 Dependency Management

Für unser Dependency Management haben wir uns aus der nativen GitHub "All-in-One"-Lösung "dependabot" bedient, damit decken wir in alle dependencies über Docker, Python und GitHub Actions alles automatisiert ab.

Listing 3.4: Bandit via GitHub Actions

```yaml
version: 2
updates:
  - package-ecosystem: "pip"
    directory: "/"
    schedule:
      interval: "weekly"
    open-pull-requests-limit: 10
    labels:
      - "dependencies"
      - "python"

  - package-ecosystem: "docker"
    directory: "/"
    schedule:
      interval: "weekly"
    open-pull-requests-limit: 5
    labels:
      - "dependencies"
      - "docker"

  - package-ecosystem: "github-actions"
    directory: "/"
    schedule:
      interval: "weekly"
    open-pull-requests-limit: 5
    labels:
      - "dependencies"
      - "github-actions"
```

3.5 Deployment

Das Deployment des Dockerimages erfolgt über eine GitHub Action bei jedem Update der Dockerfile in die Docker registry. Dort wird in der Pipeline jeweils ein Lint-Prozess und ein Build-Prozess ausgeführt, somit wird verifiziert, dass sich der Container in einem optimalen Zustand für jeden Release befindet.

Listing 3.5: Bandit via GitHub Actions

```yaml
      - name: Build and push Docker image
        uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: crnnr/weeddetector:latest
```