

Analyse Java-Abhangigkeiten

Um Ihre Anfrage zu bearbeiten, werde ich das bereitgestellte Python-Skript so ndern, dass es einen Abhangigkeitsgraph fr Java-Klassen erzeugt, wobei Abhangigkeiten nach einem bestimmten "Level" der Pakethierarchie gruppiert werden. Zum Beispiel, wenn `level=1`, werden Pakete auf der obersten Ebene gruppiert (z.B. "org"), und wenn `level=2`, werden sie auf der zweiten Ebene gruppiert (z.B. "org.springframework"). Das bedeutet, dass wir vor der Analyse der Abhangigkeiten Klassen in Gruppen basierend auf den ersten `level` Teilen ihrer Paketnamen zusammenfassen, und der Graph zeigt Abhangigkeiten zwischen diesen Gruppen an, anstatt zwischen einzelnen Klassen.

Hier ist das modifizierte Skript, gefolgt von einer Erklarung der nderungen und wie es Ihre Anforderungen erfllt.

Modifiziertes Skript

```
import os
import sys
import re
from collections import defaultdict

def get_package(file_path):
    """
    Extrahiert den Paketnamen aus einer .java-Datei.

    Args:
        file_path (str): Pfad zur .java-Datei.

    Returns:
        str: Der Paketname oder None, wenn nicht gefunden.
    """

    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                match = re.search(r'^\s*package\s+([\w.]+);', line)
                if match:
                    return match.group(1)
    except Exception as e:
        print(f"Warnung: Konnte {file_path} nicht lesen: {e}")
    return None

def get_specific_imports(file_path):
```

```
"""
```

Extrahiert spezifische Klassenimports aus einer .java-Datei, wobei Wildcard-Imports ausgeschlossen werden

Args:

file_path (str): Pfad zur .java-Datei.

Returns:

list: Liste der vollständig qualifizierten importierten Klassennamen.

```
"""
```

```
imports = []
```

```
try:
```

```
    with open(file_path, 'r', encoding='utf-8') as f:
```

```
        for line in f:
```

```
            match = re.search(r'^\s*import\s+([\w.]+);', line)
```

```
            if match:
```

```
                imp = match.group(1)
```

```
# Ausschließen von Wildcard-Imports (z.B. import java.util.*;)
```

```
                if not imp.endswith('.*'):
```

```
                    imports.append(imp)
```

```
except Exception as e:
```

```
    print(f"Warnung: Konnte {file_path} nicht lesen: {e}")
```

```
return imports
```

```
def get_package_group(full_class_name, level):
```

```
"""
```

Holt die Paketgruppe basierend auf den ersten 'level' Teilen des Paketnamens.

Args:

full_class_name (str): Vollständig qualifizierter Klassename (z.B. "org.springframework.boot.App").

level (int): Anzahl der Paketebenen, die enthalten sein sollen (z.B. 1 für "org", 2 für "org.springframework")

Returns:

str: Die Paketgruppe (z.B. "org" oder "org.springframework").

```
"""
```

```
package = '.'.join(full_class_name.split('.')[ :-1]) # Paket extrahieren, Klassennamen ausschließen
```

```
parts = package.split('.')
```

```
if len(parts) <= level:
```

```
    return package # Vollständiges Paket verwenden, wenn es weniger oder gleich viele Teile wie level hat
```

```
else:
```

```
    return '.'.join(parts[:level]) # Verwenden der ersten 'level' Teile
```

```

if __name__ == '__main__':
    # Überprüfen der Befehlszeilenargumente: root_directory und level
    if len(sys.argv) != 3:
        print("Verwendung: python script.py <root_directory> <level>")
        sys.exit(1)

    root_dir = sys.argv[1]
    try:
        level = int(sys.argv[2])
        if level < 1:
            raise ValueError
    except ValueError:
        print("Fehler: level muss eine positive ganze Zahl sein")
        sys.exit(1)

    all_classes = set()

    # Erster Durchlauf: Sammeln aller vollständig qualifizierten Klassennamen im Projekt
    for root, dirs, files in os.walk(root_dir):
        for file in files:
            if file.endswith('.java'):
                file_path = os.path.join(root, file)
                package = get_package(file_path)
                if package:
                    class_name = file.replace('.java', '')
                    full_class_name = f"{package}.{class_name}"
                    all_classes.add(full_class_name)

    # Abhängigkeiten zwischen Paketgruppen speichern
    group_dependencies = set()

    # Zweiter Durchlauf: Analysieren der Abhängigkeiten basierend auf Paketgruppen
    for root, dirs, files in os.walk(root_dir):
        for file in files:
            if file.endswith('.java'):
                file_path = os.path.join(root, file)
                package = get_package(file_path)
                if package:
                    class_name = file.replace('.java', '')

```

```

        full_class_name = f"{package}.{class_name}"
        importer_group = get_package_group(full_class_name, level)
        imports = get_specific_imports(file_path)
        for imp in imports:
            # Nur Abhängigkeiten zu Klassen innerhalb des Projekts einschließen
            # Selbstabhängigkeiten ausschließen
            if imp in all_classes and imp != full_class_name:
                imported_group = get_package_group(imp, level)
                if imported_group != importer_group:
                    group_dependencies.add((importer_group, imported_group))

# Ausgabe des Abhängigkeitsgraphen im DOT-Format
print('digraph G {')
for from_group, to_group in sorted(group_dependencies):
    print(f'    "{from_group}" -> "{to_group}";')
print('}')

```

Erklärung der Änderungen

1. Befehlszeilenargumente:

- **Original:** python script.py <root_directory>
- **Modifiziert:** python script.py <root_directory> <level>
- Unterstützung für ein zweites Argument, `level`, das die Pakethierarchieebene angibt. Das Skript überprüft, dass genau zwei Argumente bereitgestellt werden und dass `level` eine positive ganze Zahl ist.

2. Neue Funktion: `get_package_group`:

- Eine Funktion hinzugefügt, um die Paketgruppe für eine Klasse basierend auf dem angegebenen `level` zu berechnen.
- Für einen vollständig qualifizierten Klassennamen (z.B. “org.springframework.boot.App”) extrahiert es das Paket (“org.springframework.boot”), teilt es in Teile (“org”, “springframework”, “boot”) und nimmt die ersten `level` Teile:
 - Wenn `level=1`: Gibt “org” zurück.
 - Wenn `level=2`: Gibt “org.springframework” zurück.
 - Wenn das Paket weniger Teile als `level` hat (z.B. “com.example” mit `level=3`), gibt es das vollständige Paket (“com.example”) zurück.

3. Abhängigkeitsgruppierung:

- **Original:** Verwendete `defaultdict(set)`, um Abhängigkeiten zwischen einzelnen Klassen zu speichern.

- **Modifiziert:** Verwendet ein `set` (`group_dependencies`), um gerichtete Kanten zwischen Paketgruppen als Tupel (`from_group`, `to_group`) zu speichern.
- Für jede Klasse:
 - Berechnet seine Paketgruppe (`importer_group`) mit `get_package_group`.
 - Für jeden spezifischen Import, der sich innerhalb des Projekts befindet (`imp` in `all_classes`) und nicht die Klasse selbst ist (`imp != full_class_name`):
 - * Berechnet die Paketgruppe der importierten Klasse (`imported_group`).
 - * Wenn sich die Gruppen unterscheiden (`imported_group != importer_group`), fügt es eine Kante zu `group_dependencies` hinzu.
- Das `set` stellt Eindeutigkeit sicher, sodass mehrere Abhängigkeiten zwischen denselben Gruppen zu einer einzigen Kante führen.

4. DOT-Ausgabe:

- **Original:** Druckte Kanten zwischen einzelnen Klassen (z.B. “org.springframework.boot.App”-> “org.apache.commons.IOUtils”).
- **Modifiziert:** Druckt Kanten zwischen Paketgruppen (z.B. “org.springframework”-> “org.apache” für `level=2`).
- Kanten werden sortiert, um eine konsistente Ausgabe zu gewährleisten.

Wie es Ihre Anforderungen erfüllt

- **Unterstützung für Ebenen:** Das Skript akzeptiert jetzt einen `level`-Parameter, um Pakete vor der Analyse der Abhängigkeiten zu gruppieren.
- **Ebene = 1:** Gruppert alle Klassen nach ihrem obersten Paket (z.B. “org”). Zum Beispiel gehören “org.springframework.boot.App” und “org.apache.commons.IOUtils” beide zur “org”-Gruppe, sodass Importe zwischen ihnen innerhalb von “org” nicht als Kanten angezeigt werden.
- **Ebene = 2:** Gruppert Klassen nach den ersten zwei Paketebenen (z.B. “org.springframework”). Zum Beispiel erstellt ein Import von “org.springframework.boot.App” zu “org.apache.commons.IOUtils” eine Kante von “org.springframework” zu “org.apache”.
- **Pakete vor der Abhängigkeitsanalyse akkumulieren:** Das Skript bestimmt die Paketgruppe jeder Klasse basierend auf `level` vor der Analyse ihrer Importe, sodass Abhängigkeiten zwischen Gruppen und nicht zwischen einzelnen Klassen bestehen.
- **Beispielkonformität:** Für Ihre Beispielkante:
 - Original: “org.springframework.boot.web.servlet.server.Session” -> “org.springframework.boot.convert.D
 - Mit `level=2`: Beide Klassen befinden sich in “org.springframework”, sodass keine Kante hinzugefügt wird (gleiche Gruppe).
 - Mit `level=3`: “org.springframework.boot.web”-> “org.springframework.boot.convert”, fügt eine Kante zwischen diesen verschiedenen Gruppen hinzu.

Verwendungsbeispiele

- **Befehl:** `python script.py /pfad/zum/projekt 1`
 - Gruppen: "org", "com", usw.
 - Ausgabe könnte enthalten: "org" -> "com", wenn eine Klasse in einem "org"-Paket eine Klasse in einem "com"-Paket importiert.
- **Befehl:** `python script.py /pfad/zum/projekt 2`
 - Gruppen: "org.springframework", "org.apache", usw.
 - Ausgabe könnte enthalten: "org.springframework" -> "org.apache".

Hinweise

- Das Skript behält das ursprüngliche Verhalten bei, nur Abhängigkeiten innerhalb des Projekts (Imports innerhalb von `all_classes`) einzuschließen und Wildcard-Imports und Selbstabhängigkeiten auszuschließen.
- Klassen ohne Paketdeklaration werden übersprungen, was mit dem ursprünglichen Skript übereinstimmt.
- Die Ausgabe ist ein gerichteter Graph im DOT-Format, der mit Tools wie Graphviz angezeigt werden kann.

Dieses modifizierte Skript sollte Ihre Anfrage vollständig erfüllen, Abhängigkeiten nach Pakethierarchieebene zu gruppieren.