Analyser les dépendances Java

Pour répondre à votre demande, je vais modifier le script Python fourni pour générer un graphe de dépendances pour les classes Java, où les dépendances sont regroupées par un niveau spécifié de la hiérarchie des packages. Par exemple, si `level=1`, les packages sont regroupés au niveau supérieur (par exemple, “org”), et si `level=2`, ils sont regroupés au deuxième niveau (par exemple, “org.springframework”). Cela signifie qu’avant d’analyser les dépendances, nous accumulons les classes en groupes en fonction des premières parties de niveau de leurs noms de packages, et le graphe montrera les dépendances entre ces groupes plutôt qu’entre des classes individuelles.

Voici le script modifié, suivi d’une explication des modifications et de la manière dont il répond à vos exigences.

Script Modifié

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

def get_package(file_path):
    """
    Extraire le nom du package à partir d'un fichier .java.

    Args:
        file_path (str): Chemin vers le fichier .java.

    Returns:
        str: Le nom du package, ou None s'il n'est pas trouvé.
    """
    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"Attention: Impossible de lire {file_path}: {e}")
    return None

def get_specific_imports(file_path):
```

```
    """
    Extraire les importations de classes spécifiques à partir d'un fichier .java, en excluant les importations

    Args:
        file_path (str): Chemin vers le fichier .java.

    Returns:
        list: Liste des noms de classes importées qualifiés.
    """
    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)
                    # Exclure les importations génériques (par exemple, import java.util.*;)
                    if not imp.endswith('.*'):
                        imports.append(imp)
    except Exception as e:
        print(f"Attention: Impossible de lire {file_path}: {e}")
    return imports

def get_package_group(full_class_name, level):
    """
    Obtenir le groupe de packages en fonction des premières parties de niveau du nom du package.

    Args:
        full_class_name (str): Nom de classe qualifié (par exemple, "org.springframework.boot.App").
        level (int): Nombre de niveaux de packages à inclure (par exemple, 1 pour "org", 2 pour "org.springfr

    Returns:
        str: Le groupe de packages (par exemple, "org" ou "org.springframework").
    """
    package = '.'.join(full_class_name.split('.')[:-1])  # Extraire le package, en excluant le nom de la class
    parts = package.split('.')
    if len(parts) <= level:
        return package  # Utiliser le package complet s'il a moins ou autant de parties que le niveau
    else:
        return '.'.join(parts[:level])  # Utiliser les premières parties de niveau
```

```python
if __name__ == '__main__':
    # Vérifier les arguments de la ligne de commande: root_directory et level
    if len(sys.argv) != 3:
        print("Utilisation: 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("Erreur: level doit être un entier positif")
        sys.exit(1)

    all_classes = set()

    # Premier passage: Collecter tous les noms de classes qualifiés dans le projet
    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)

    # Stocker les dépendances entre les groupes de packages
    group_dependencies = set()

    # Deuxième passage: Analyser les dépendances en fonction des groupes de packages
    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:
                    # Inclure uniquement les dépendances sur les classes au sein du projet
                    # Exclure les dépendances sur soi-même
                    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))

# Afficher le graphe de dépendances au format DOT
print('digraph G {')
for from_group, to_group in sorted(group_dependencies):
    print(f'  "{from_group}" -> "{to_group}";')
print('}')
```

Explication des Modifications

1. **Arguments de la Ligne de Commande**:
   - **Original**: `python script.py <root_directory>`
   - **Modifié**: `python script.py <root_directory> <level>`
   - Ajout du support pour un deuxième argument, `level`, qui spécifie le niveau de la hiérarchie des packages. Le script vérifie que exactement deux arguments sont fournis et que `level` est un entier positif.
2. **Nouvelle Fonction:** `get_package_group`:
   - Ajout d'une fonction pour calculer le groupe de packages pour une classe en fonction du `level` spécifié.
   - Pour un nom de classe qualifié (par exemple, "org.springframework.boot.App"), il extrait le package ("org.springframework.boot"), le divise en parties ("org", "springframework", "boot"), et prend les premières parties de niveau:
     - Si `level=1`: Retourne "org".
     - Si `level=2`: Retourne "org.springframework".
     - Si le package a moins de parties que `level` (par exemple, "com.example"avec `level=3`), il retourne le package complet ("com.example").
3. **Groupement des Dépendances**:
   - **Original**: Utilisait `defaultdict(set)` pour stocker les dépendances entre des classes individuelles.
   - **Modifié**: Utilise un `set` (`group_dependencies`) pour stocker les arêtes dirigées entre les groupes de packages sous forme de tuples `(from_group, to_group)`.

- Pour chaque classe:
  - Calcule son groupe de packages (`importer_group`) en utilisant `get_package_group`.
  - Pour chaque importation spécifique qui est au sein du projet (`imp in all_classes`) et n'est pas la classe elle-même (`imp != full_class_name`):
    * Calcule le groupe de packages de la classe importée (`imported_group`).
    * Si les groupes diffèrent (`imported_group != importer_group`), ajoute une arête à `group_dependencies`.
- Le `set` assure l'unicité, donc plusieurs dépendances entre les mêmes groupes résultent en une seule arête.

4. **Sortie DOT**:
   - **Original**: Imprimait les arêtes entre des classes individuelles (par exemple, "org.springframework.boot.Ap -> "org.apache.commons.IOUtils").
   - **Modifié**: Imprime les arêtes entre les groupes de packages (par exemple, "org.springframework" -> "org.apache"pour `level=2`).
   - Les arêtes sont triées pour une sortie cohérente.

Comment Cela Répond à Vos Exigences

- **Support pour les Niveaux**: Le script accepte maintenant un paramètre `level` pour regrouper les packages avant d'analyser les dépendances.
- **Niveau = 1**: Regroupe toutes les classes par leur package de niveau supérieur (par exemple, "org"). Par exemple, "org.springframework.boot.App"et "org.apache.commons.IOUtils"appartiennent tous deux au groupe "org", donc les importations entre eux au sein de "org"ne sont pas montrées comme des arêtes.
- **Niveau = 2**: Regroupe les classes par les deux premiers niveaux de packages (par exemple, "org.springframework"). Par exemple, une importation de "org.springframework.boot.App"vers "org.apache.commons.IOUtils"crée une arête de "org.springframework"vers "org.apache".
- **Accumuler les Packages Avant l'Analyse des Dépendances**: Le script détermine le groupe de packages de chaque classe en fonction de `level` avant d'analyser ses importations, assurant ainsi que les dépendances sont entre les groupes, pas entre des classes individuelles.
- **Exemple de Conformité**: Pour votre exemple d'arête:
  - Original: `"org.springframework.boot.web.servlet.server.Session" -> "org.springframework.boot.convert.D`
  - Avec `level=2`: Les deux classes sont dans "org.springframework", donc aucune arête n'est ajoutée (même groupe).
  - Avec `level=3`: "org.springframework.boot.web"-> "org.springframework.boot.convert", ajoutant une arête entre ces groupes distincts.

Exemples d'Utilisation

- **Commande**: `python script.py /chemin/vers/projet 1`

  - Groupes: “org”, “com”, etc.
  - La sortie pourrait inclure: `"org" -> "com"` si une classe dans un package “org”importe une classe dans un package “com”.

- **Commande**: `python script.py /chemin/vers/projet 2`
  - Groupes: “org.springframework”, “org.apache”, etc.
  - La sortie pourrait inclure: `"org.springframework" -> "org.apache"`.

Notes

- Le script conserve le comportement original d’inclure uniquement les dépendances intra-projet (importations au sein de `all_classes`) et d’exclure les importations génériques et les dépendances sur soi-même.
- Les classes sans déclaration de package sont ignorées, de manière cohérente avec le script original.
- La sortie est un graphe dirigé au format DOT, visualisable avec des outils comme Graphviz.

Ce script modifié devrait pleinement satisfaire votre demande de regrouper les dépendances par niveau de hiérarchie des packages.