Analyser les POM Maven pour Générer un Graphique de Dépendances

Pour répondre à votre demande, je vais fournir une solution pour générer un graphe de dépendances au format DOT en trouvant de manière récursive tous les fichiers `pom.xml` dans un répertoire et en analysant leurs dépendances, sans supposer une structure de projet Maven multi-module fixe comme celle de l'exemple (`project/module1/pom.xml`, etc.). Le script se concentrera d'abord sur la collecte de tous les fichiers `pom.xml`, puis sur leur analyse pour déterminer les dépendances des modules et générer le graphe.

Vue d'ensemble de la solution

Voici ce que le script va faire : 1. **Collecter récursivement les fichiers** `pom.xml` : Parcourir le répertoire racine spécifié pour trouver tous les fichiers `pom.xml`, quelle que soit leur localisation ou structure. 2. **Analyser les informations des modules** : Extraire `groupId`, `artifactId` et les dépendances de chaque `pom.xml`. 3. **Gérer l'héritage de** `groupId` : Si un `pom.xml` ne spécifie pas de `groupId`, l'hériter d'une déclaration `<parent>`, si elle est présente. 4. **Identifier les dépendances intra-projet** : Correspondre les dépendances aux modules au sein du projet en fonction de `groupId` et `artifactId`. 5. **Générer la sortie DOT** : Créer un graphe orienté des dépendances entre les modules.

Cette approche ne repose pas sur une hiérarchie de répertoires prédéfinie ni sur l'hypothèse que les fichiers `pom.xml` parents se trouvent dans des répertoires parents.

Script Python

Voici le script Python adapté à vos besoins :

```python
import os
import sys
import xml.etree.ElementTree as ET

# Définir l'espace de noms Maven POM
NS = "{http://maven.apache.org/POM/4.0.0}"

# Cache pour groupId pour éviter un parsing redondant
group_id_cache = {}
```

```python
def get_group_id(pom_path, pom_map):
    """
    Extraire le groupId d'un fichier pom.xml, en tenant compte de l'héritage à partir d'un parent.

    Args:
        pom_path (str): Chemin vers le fichier pom.xml.
        pom_map (dict): Mappage des chemins des fichiers pom.xml à leurs données analysées.

    Returns:
        str: Le groupId du module.
    """
    if pom_path in group_id_cache:
        return group_id_cache[pom_path]

    tree = ET.parse(pom_path)
    root = tree.getroot()
    group_id_elem = root.find(NS + 'groupId')

    if group_id_elem is not None:
        group_id = group_id_elem.text.strip()
    else:
        # Vérifier la déclaration du parent
        parent = root.find(NS + 'parent')
        if parent is not None:
            parent_group_id = parent.find(NS + 'groupId').text.strip()
            parent_artifact_id = parent.find(NS + 'artifactId').text.strip()
            parent_relative_path = parent.find(NS + 'relativePath')
            if parent_relative_path is not None and parent_relative_path.text:
                parent_pom_path = os.path.normpath(
                    os.path.join(os.path.dirname(pom_path), parent_relative_path.text)
                )
            else:
                # Par défaut, utiliser le répertoire parent si relativePath est omis
                parent_pom_path = os.path.join(os.path.dirname(pom_path), '..', 'pom.xml')
                parent_pom_path = os.path.normpath(parent_pom_path)

            if parent_pom_path in pom_map:
                group_id = get_group_id(parent_pom_path, pom_map)
            else:
                raise ValueError(f"Parent POM non trouvé pour {pom_path}: {parent_pom_path}")
```

```
        else:
            raise ValueError(f"Aucun groupId ou parent spécifié dans {pom_path}")

    group_id_cache[pom_path] = group_id
    return group_id

def get_artifact_id(pom_path):
    """
    Extraire l'artifactId d'un fichier pom.xml.

    Args:
        pom_path (str): Chemin vers le fichier pom.xml.

    Returns:
        str: L'artifactId du module.
    """
    tree = ET.parse(pom_path)
    root = tree.getroot()
    artifact_id_elem = root.find(NS + 'artifactId')

    if artifact_id_elem is None:
        raise ValueError(f"Le pom.xml doit spécifier artifactId: {pom_path}")

    return artifact_id_elem.text.strip()

def get_dependencies(pom_path):
    """
    Extraire la liste des dépendances d'un fichier pom.xml.

    Args:
        pom_path (str): Chemin vers le fichier pom.xml.

    Returns:
        list: Liste de tuples (groupId, artifactId) pour chaque dépendance.
    """
    tree = ET.parse(pom_path)
    root = tree.getroot()
    dependencies = []

    for dep in root.findall(NS + 'dependencies/' + NS + 'dependency'):
```

```
            dep_group_id_elem = dep.find(NS + 'groupId')
            dep_artifact_id_elem = dep.find(NS + 'artifactId')
            if dep_group_id_elem is not None and dep_artifact_id_elem is not None:
                dep_group_id = dep_group_id_elem.text.strip()
                dep_artifact_id = dep_artifact_id_elem.text.strip()
                dependencies.append((dep_group_id, dep_artifact_id))

    return dependencies

if __name__ == '__main__':
    # Vérifier les arguments de la ligne de commande
    if len(sys.argv) != 2:
        print("Usage: python script.py <répertoire_racine>")
        sys.exit(1)

    root_dir = sys.argv[1]
    if not os.path.isdir(root_dir):
        print(f"Erreur: {root_dir} n'est pas un répertoire")
        sys.exit(1)

    # Étape 1 : Trouver récursivement tous les fichiers pom.xml
    pom_files = [
        os.path.join(root, file)
        for root, _, files in os.walk(root_dir)
        for file in files if file == 'pom.xml'
    ]

    if not pom_files:
        print(f"Aucun fichier pom.xml trouvé dans {root_dir}")
        sys.exit(1)

    # Étape 2 : Construire un dictionnaire de tous les POM pour les recherches de parent
    pom_map = {pom_file: None for pom_file in pom_files}

    # Étape 3 : Extraire les informations des modules
    modules = {}  # (groupId, artifactId) -> chemin_pom
    for pom_file in pom_files:
        try:
            group_id = get_group_id(pom_file, pom_map)
            artifact_id = get_artifact_id(pom_file)
```

```
            modules[(group_id, artifact_id)] = pom_file
        except ValueError as e:
            print(f"Avertissement : Ignorer {pom_file} en raison de l'erreur : {e}")
            continue

    # Étape 4 : Analyser les dépendances
    dependencies = set()
    for pom_file in pom_files:
        try:
            importer_group_id = get_group_id(pom_file, pom_map)
            importer_artifact_id = get_artifact_id(pom_file)
            importer_key = (importer_group_id, importer_artifact_id)
            deps = get_dependencies(pom_file)
            for dep_group_id, dep_artifact_id in deps:
                dep_key = (dep_group_id, dep_artifact_id)
                if dep_key in modules and dep_key != importer_key:
                    # Ajouter la dépendance comme un tuple (importer, importé) en utilisant artifactId pour la
                    dependencies.add((importer_artifact_id, dep_artifact_id))
        except ValueError as e:
            print(f"Avertissement : Erreur lors du traitement des dépendances dans {pom_file} : {e}")
            continue

    # Étape 5 : Sortie au format DOT
    print('digraph G {')
    for from_module, to_module in sorted(dependencies):
        print(f'  "{from_module}" -> "{to_module}";')
    print('}')
```

Fonctionnement

1. Entrée en ligne de commande

- Prend un seul argument : `<répertoire_racine>`, le point de départ pour la recherche récursive.
- Valide qu'il s'agit d'un répertoire.

2. Recherche des fichiers `pom.xml`

- Utilise `os.walk` pour parcourir récursivement l'arborescence des répertoires et collecter tous les fichiers `pom.xml` dans une liste.

3. Analyse des informations des modules

- **Group ID (`groupId`)** :
  - Extrait de chaque `pom.xml`.
  - Si non présent, recherche une section `<parent>` et résout le `groupId` à partir du POM parent référencé en utilisant `relativePath` (ou par défaut au répertoire parent si omis).
  - Met en cache les résultats pour éviter un re-parsing.
- **Artifact ID (`artifactId`)**: Doit être présent dans chaque `pom.xml`.
- **Dépendances**: Extrait les paires `(groupId, artifactId)` de la section `<dependencies>`.

4. Analyse des dépendances

- Construit une carte de `(groupId, artifactId)` vers `chemin_pom` pour tous les modules.
- Pour chaque `pom.xml`, vérifie ses dépendances contre la carte des modules pour trouver les dépendances intra-projet.
- Exclut les dépendances de soi-même (où un module dépend de lui-même).
- Enregistre les dépendances comme des paires `(importer_artifactId, imported_artifactId)`.

5. Sortie DOT

- Génère un graphe orienté au format DOT, en utilisant `artifactId` comme étiquettes de nœuds pour la simplicité.

Exemple d'utilisation

Supposons que votre structure de répertoire soit irrégulière :

```
myproject/
  app/pom.xml (groupId="com.myapp", artifactId="app", dépend de "core")
  libs/core/pom.xml (groupId="com.myapp", artifactId="core")
  tools/util/pom.xml (groupId="com.myapp", artifactId="util", dépend de "core")
```

Exécutez le script :

```
python script.py myproject
```

Sortie :

```
digraph G {
  "app" -> "core";
  "util" -> "core";
}
```

Cela montre que `app` et `util` dépendent tous deux de `core`.

Notes et limitations

- **Pas de regroupement par répertoire** : Contrairement au script original dans la trace de pensée, cette version ne regroupe pas les modules par structure de répertoire puisque vous ne l'avez pas demandé. Elle utilise `artifactId` directement dans le graphe.
- **Résolution des parents** : S'appuie sur `relativePath` dans la section `<parent>` ou suppose que le POM parent se trouve dans le répertoire parent si omis. Si le POM parent n'est pas dans l'ensemble collecté, il lève une erreur.
- **Correspondance simple** : Correspond les dépendances par `groupId` et `artifactId` uniquement, ignorant les versions.
- **Pas de substitution de propriétés** : Ne gère pas les propriétés Maven comme `${project.groupId}`.
- **Gestion des erreurs** : Ignore les fichiers `pom.xml` problématiques avec un avertissement au lieu de tout arrêter.

Ce script répond à votre exigence de "parcourir récursivement, obtenir tous les POMs d'abord, puis analyser" sans supposer une structure de fichiers fixe. Faites-moi savoir si vous avez besoin d'ajustements !