Analyse Maven POMs zur Erstellung eines Abhängigkeitsgraphen

Um Ihre Anfrage zu beantworten, werde ich eine Lösung zur Erstellung eines Abhängigkeitsgraphen im DOT-Format bereitstellen, indem alle `pom.xml`-Dateien in einem Verzeichnis rekursiv gesucht und deren Abhängigkeiten analysiert werden, ohne eine feste Maven-Multimodul-Projektstruktur wie im Beispiel (`project/module1/pom.xml` usw.) vorauszusetzen. Das Skript wird sich darauf konzentrieren, zunächst alle `pom.xml`-Dateien zu sammeln, diese dann zu analysieren, um Modulabhängigkeiten zu bestimmen und den Graphen auszugeben.

Lösung

Hier ist, was das Skript tun wird: 1. **Rekursives Sammeln von `pom.xml`-Dateien**: Durchsuchen Sie das angegebene Stammverzeichnis, um alle `pom.xml`-Dateien zu finden, unabhängig von ihrem Standort oder ihrer Struktur. 2. **Parsen von Modulinformationen**: Extrahieren Sie `groupId`, `artifactId` und Abhängigkeiten aus jeder `pom.xml`. 3. **Vererben von `groupId`**: Wenn eine `pom.xml` kein `groupId` angibt, erben Sie es von einer `<parent>`-Angabe, falls vorhanden. 4. **Identifizieren von Intra-Projekt-Abhängigkeiten**: Abhängigkeiten innerhalb des Projekts basierend auf `groupId` und `artifactId` zuordnen. 5. **Erstellen der DOT-Ausgabe**: Erstellen Sie einen gerichteten Graphen der Abhängigkeiten zwischen Modulen.

Diese Vorgehensweise setzt keine vordefinierte Verzeichnisstruktur voraus oder geht davon aus, dass übergeordnete `pom.xml`-Dateien in übergeordneten Verzeichnissen liegen.

Python-Skript

Hier ist das Python-Skript, das auf Ihre Anforderungen zugeschnitten ist:

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

# Definieren des Maven POM-Namensraums
NS = "{http://maven.apache.org/POM/4.0.0}"

# Cache für groupId, um redundantes Parsen zu vermeiden
group_id_cache = {}

def get_group_id(pom_path, pom_map):
```

```
    """
    Extrahieren Sie die groupId aus einer pom.xml-Datei, wobei die Vererbung von einem Elternteil berücksicht

    Args:
        pom_path (str): Pfad zur pom.xml-Datei.
        pom_map (dict): Zuordnung von pom.xml-Pfaden zu ihren geparsten Daten.

    Returns:
        str: Die groupId des Moduls.
    """
    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:
        # Überprüfen Sie auf eine Elternangabe
        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:
                # Standardmäßig auf das übergeordnete Verzeichnis, wenn relativePath weggelassen wird
                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"Eltern-POM nicht gefunden für {pom_path}: {parent_pom_path}")
        else:
```

```
            raise ValueError(f"Keine groupId oder Elternangabe in {pom_path} angegeben")

    group_id_cache[pom_path] = group_id
    return group_id

def get_artifact_id(pom_path):
    """
    Extrahieren Sie die artifactId aus einer pom.xml-Datei.

    Args:
        pom_path (str): Pfad zur pom.xml-Datei.

    Returns:
        str: Die artifactId des Moduls.
    """
    tree = ET.parse(pom_path)
    root = tree.getroot()
    artifact_id_elem = root.find(NS + 'artifactId')

    if artifact_id_elem is None:
        raise ValueError(f"pom.xml muss artifactId angeben: {pom_path}")

    return artifact_id_elem.text.strip()

def get_dependencies(pom_path):
    """
    Extrahieren Sie die Liste der Abhängigkeiten aus einer pom.xml-Datei.

    Args:
        pom_path (str): Pfad zur pom.xml-Datei.

    Returns:
        list: Liste von Tupeln (groupId, artifactId) für jede Abhängigkeit.
    """
    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__':
    # Überprüfen Sie die Befehlszeilenargumente
    if len(sys.argv) != 2:
        print("Verwendung: python script.py <root_directory>")
        sys.exit(1)

    root_dir = sys.argv[1]
    if not os.path.isdir(root_dir):
        print(f"Fehler: {root_dir} ist kein Verzeichnis")
        sys.exit(1)

    # Schritt 1: Rekursives Finden aller pom.xml-Dateien
    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"Keine pom.xml-Dateien in {root_dir} gefunden")
        sys.exit(1)

    # Schritt 2: Erstellen Sie ein Wörterbuch aller POMs für Elternabfragen
    pom_map = {pom_file: None for pom_file in pom_files}

    # Schritt 3: Extrahieren Sie Modulinformationen
    modules = {}  # (groupId, artifactId) -> pom_path
    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"Warnung: Überspringen von {pom_file} aufgrund eines Fehlers: {e}")
            continue

    # Schritt 4: Analysieren Sie Abhängigkeiten
    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:
                    # Fügen Sie die Abhängigkeit als (Importer, Importiert) Tupel hinzu, wobei artifactId für
                    dependencies.add((importer_artifact_id, dep_artifact_id))
        except ValueError as e:
            print(f"Warnung: Fehler beim Verarbeiten der Abhängigkeiten in {pom_file}: {e}")
            continue

    # Schritt 5: Ausgabe im DOT-Format
    print('digraph G {')
    for from_module, to_module in sorted(dependencies):
        print(f'  "{from_module}" -> "{to_module}";')
    print('}')
```

Funktionsweise

1. Befehlszeileneingabe

- Nimmt ein einziges Argument entgegen: `<root_directory>`, der Ausgangspunkt für die rekursive Suche.
- Validiert, dass es sich um ein Verzeichnis handelt.

2. Finden von `pom.xml`-Dateien

- Verwendet `os.walk`, um rekursiv das Verzeichnisbaum zu durchsuchen und alle `pom.xml`-Dateien in eine Liste zu sammeln.

3. Parsen von Modulinformationen

- **Gruppen-ID (`groupId`)**:
  - Wird aus jeder `pom.xml` extrahiert.
  - Wenn nicht vorhanden, wird nach einer `<parent>`-Angabe gesucht und die `groupId` aus dem referenzierten Eltern-POM aufgelöst, wobei der `relativePath` verwendet wird (oder standardmäßig auf das übergeordnete Verzeichnis, wenn weggelassen).
  - Zwischenspeichert Ergebnisse, um redundantes Parsen zu vermeiden.
- **Artifact-ID (`artifactId`)**: Muss in jeder `pom.xml` vorhanden sein.
- **Abhängigkeiten**: Extrahiert `(groupId, artifactId)`-Paare aus dem `<dependencies>`-Bereich.

4. Abhängigkeitsanalyse

- Erstellt eine Zuordnung von `(groupId, artifactId)` zu `pom_path` für alle Module.
- Für jede `pom.xml` prüft es deren Abhängigkeiten gegen die Modulzuordnung, um Intra-Projekt-Abhängigkeiten zu finden.
- Ausschließen von Selbstabhängigkeiten (wenn ein Modul von sich selbst abhängt).
- Abhängigkeiten als `(importer_artifactId, imported_artifactId)`-Paare aufzeichnen.

5. DOT-Ausgabe

- Gibt einen gerichteten Graphen im DOT-Format aus, wobei `artifactId` als Knotenbezeichnungen verwendet wird.

Beispielverwendung

Angenommen, Ihre Verzeichnisstruktur ist unregelmäßig:

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

Führen Sie das Skript aus:

```
python script.py myproject
```

Ausgabe:

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

Dies zeigt, dass sowohl `app` als auch `util` von `core` abhängen.

Hinweise und Einschränkungen

- **Keine Gruppierung nach Verzeichnis**: Im Gegensatz zum ursprünglichen Skript in der Denkspur gruppiert diese Version die Module nicht nach Verzeichnisstruktur, da Sie dies nicht angefordert haben. Sie verwendet `artifactId` direkt im Graphen.
- **Elternauflösung**: Verwendet `relativePath` im `<parent>`-Bereich oder geht davon aus, dass das Eltern-POM im übergeordneten Verzeichnis liegt, wenn weggelassen. Wenn das Eltern-POM nicht in der gesammelten Menge vorhanden ist, wird ein Fehler ausgelöst.
- **Einfache Übereinstimmung**: Übereinstimmung von Abhängigkeiten nur anhand von `groupId` und `artifactId`, Versionen werden ignoriert.
- **Keine Eigenschaftssubstitution**: Behandelt keine Maven-Eigenschaften wie `${project.groupId}`.
- **Fehlerbehandlung**: Überspringt problematische `pom.xml`-Dateien mit einer Warnung, anstatt vollständig zu scheitern.

Dieses Skript erfüllt Ihre Anforderung, „rekursiv zu iterieren, alle POMs zuerst zu erhalten und dann zu analysieren“, ohne eine feste Dateistruktur vorauszusetzen. Lassen Sie mich wissen, wenn Sie Anpassungen benötigen!