Analyze Java Dependencies

To address your query, I'll modify the provided Python script to generate a dependency graph for Java classes, where dependencies are grouped by a specified "level"of the package hierarchy. For example, if `level=1`, packages are grouped at the top level (e.g., "org"), and if `level=2`, they are grouped at the second level (e.g., "org.springframework"). This means that before analyzing dependencies, we accumulate classes into groups based on the first `level` parts of their package names, and the graph will show dependencies between these groups rather than individual classes.

Below is the modified script, followed by an explanation of the changes and how it meets your requirements.

Modified Script

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

def get_package(file_path):
    """
    Extract the package name from a .java file.

    Args:
        file_path (str): Path to the .java file.

    Returns:
        str: The package name, or None if not found.
    """
    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"Warning: Could not read {file_path}: {e}")
    return None

def get_specific_imports(file_path):
    """
    Extract specific class imports from a .java file, excluding wildcard imports.
    Args:
        file_path (str): Path to the .java file.

    Returns:
        list: List of fully qualified imported class names.
    """
    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)
                    # Exclude wildcard imports (e.g., import java.util.*;)
                    if not imp.endswith('.*'):
                        imports.append(imp)
    except Exception as e:
        print(f"Warning: Could not read {file_path}: {e}")
    return imports

def get_package_group(full_class_name, level):
    """
    Get the package group based on the first 'level' parts of the package name.

    Args:
        full_class_name (str): Fully qualified class name (e.g., "org.springframework.boot.App").
        level (int): Number of package levels to include (e.g., 1 for "org", 2 for "org.springframework").

    Returns:
        str: The package group (e.g., "org" or "org.springframework").
    """
    package = '.'.join(full_class_name.split('.')[:-1])  # Extract package, excluding class name
    parts = package.split('.')
    if len(parts) <= level:
        return package  # Use full package if it has fewer or equal parts than level
    else:
        return '.'.join(parts[:level])  # Use first 'level' parts

if __name__ == '__main__':
```

```python
# Check for command-line arguments: root_directory and level
if len(sys.argv) != 3:
    print("Usage: 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("Error: level must be a positive integer")
    sys.exit(1)

all_classes = set()

# First pass: Collect all fully qualified class names in the project
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)

# Store dependencies between package groups
group_dependencies = set()

# Second pass: Analyze dependencies based on package groups
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:
                    # Only include dependencies on classes within the project
                    # Exclude self-dependencies
                    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))

# Output the dependency graph in DOT format
print('digraph G {')
for from_group, to_group in sorted(group_dependencies):
    print(f'  "{from_group}" -> "{to_group}";')
print('}')
```

Explanation of Changes

1. **Command-Line Arguments**:
   - **Original**: `python script.py <root_directory>`
   - **Modified**: `python script.py <root_directory> <level>`
   - Added support for a second argument, `level`, which specifies the package hierarchy level. The script checks that exactly two arguments are provided and that `level` is a positive integer.
2. **New Function:** `get_package_group`:
   - Added a function to compute the package group for a class based on the specified `level`.
   - For a fully qualified class name (e.g., “org.springframework.boot.App”), it extracts the package (“org.springframework.boot”), splits it into parts (“org”, “springframework”, “boot”), and takes the first `level` parts:
     - If `level=1`: Returns “org”.
     - If `level=2`: Returns “org.springframework”.
     - If the package has fewer parts than `level` (e.g., “com.example”with `level=3`), it returns the full package (“com.example”).
3. **Dependency Grouping**:
   - **Original**: Used `defaultdict(set)` to store dependencies between individual classes.
   - **Modified**: Uses a `set` (`group_dependencies`) to store directed edges between package groups as tuples `(from_group, to_group)`.
   - For each class:
     - Computes its package group (`importer_group`) using `get_package_group`.
     - For each specific import that is within the project (`imp in all_classes`) and not the class itself (`imp != full_class_name`):

        * Computes the imported class's package group (`imported_group`).
        * If the groups differ (`imported_group != importer_group`), adds an edge to `group_dependencies`.
   - The `set` ensures uniqueness, so multiple dependencies between the same groups result in a single edge.

4. **DOT Output**:
   - **Original**: Printed edges between individual classes (e.g., "org.springframework.boot.App"-> "org.apache.commons.IOUtils").
   - **Modified**: Prints edges between package groups (e.g., "org.springframework"-> "org.apache"for `level=2`).
   - Edges are sorted for consistent output.

How It Meets Your Requirements

- **Support for Levels**: The script now accepts a `level` parameter to group packages before analyzing dependencies.
- **Level = 1**: Groups all classes by their top-level package (e.g., "org"). For example, "org.springframework.boot. and "org.apache.commons.IOUtils"both belong to the "org"group, so imports between them within "org"are not shown as edges.
- **Level = 2**: Groups classes by the first two package levels (e.g., "org.springframework"). For example, an import from "org.springframework.boot.App"to "org.apache.commons.IOUtils"creates an edge from "org.springframework"to "org.apache".
- **Accumulate Packages Before Dependency Analysis**: The script determines each class's package group based on `level` before analyzing its imports, ensuring dependencies are between groups, not individual classes.
- **Example Compliance**: For your example edge:
  - Original: `"org.springframework.boot.web.servlet.server.Session" -> "org.springframework.boot.convert.D`
  - With `level=2`: Both classes are in "org.springframework", so no edge is added (same group).
  - With `level=3`: "org.springframework.boot.web"-> "org.springframework.boot.convert", adding an edge between these distinct groups.

Usage Examples

- **Command**: `python script.py /path/to/project 1`
  - Groups: "org", "com", etc.
  - Output might include: `"org" -> "com"` if a class in an "org"package imports a class in a "com" package.
- **Command**: `python script.py /path/to/project 2`
  - Groups: "org.springframework", "org.apache", etc.
  - Output might include: `"org.springframework" -> "org.apache"`.

Notes

- The script maintains the original behavior of only including intra-project dependencies (imports within `all_classes`) and excluding wildcard imports and self-dependencies.
- Classes without a package declaration are skipped, consistent with the original script.
- The output is a directed graph in DOT format, viewable with tools like Graphviz.

This modified script should fully satisfy your request to group dependencies by package level hierarchy.