DON Dependency Resolution

Unlike `DPM`, `DON` handles most aspects of dependency resolution.

The algorithm used for dependency resolution in DON is a modified breadth-first traversal of a directed package dependency graph to detect cycles while building an installation-ordered package list between packages available in its repositories, and packages already installed.

The dependency graph starts as a set of flat associated key/pair set compiled from all the repos available on the system, and already installed on the system itself. This is the association between package NAME|PROVIDES with PACKAGE_MANIFEST checksum and PACKAGE_MANIFEST checksum with DEPENDS declaration rules in the DPM Repository Metadata database that gets cached for the repo locally by DON.

That's alot to take in. In other terms:

1. DON downloads the metadata.db from every repository it's configured to use. Each cached database for each repository contains the packages in the repository, their versions, and those packages' dependencies (among other things not relevant to this section).
2. When DON is told to install a package, it takes the name, finds the most recent version of the package with that NAME or PROVIDES declaration from its repositories, picking the first one it sees according to the weight of the repositories (as defined in each repository configuration file in `/etc/don.repos.d/`).
3. DON will then run a method that finds the DEPENDENCIES declared in that package and evaluates each dependency rule against packages already installed, and packages available in any repository it speaks to. As it does this, for any packages not already installed, it checks the dependencies declared for those packages in their corresponding repository metadata.db in an iterative fashion. If it detects circular dependencies, it will fail unless an override switch is supplied. So in this way, DON is walking through the dependencies declared for the package it is told to install, and then checking the dependencies for those packages, until it has a full set of packages required to be installed, with

logic to detect circular dependency declaration and a method to override it.
4. DON will then order the packages in the order they need to be installed to meet all the dependencies and begin installation.

Educational Demonstration

If this is still confusing, here is a very rudimentary implementation in python, that serves purely an educational purpose to illustrate this concept:

```
class Package:
    def __init__( self, package_name, package_version )
        pass

    # returns a list of Package objects
    def get_dependencies()
        pass

def main():
    package_name = "httpd"
     # Final ordered list of all packages that need to be
installed
    final_packages_list = []
    # Queue of packages we still need to process
    packages_to_check = []
     # Running list of current dependency chain to detect
circular dependencies
    dep_chain = [ package_name ]

    start_pkg = Package( package_name, "1.0" )
    packages_to_check = [start_pkg]
    for pkg in packages_to_check:
        # Check if we've seen this before in the chain
        if pkg.name in dep_chain[ :-1 ]:
             raise Exception( f"Circular dependency detected:
{pkg.name}" )
        # Already processed this package
        if pkg in final_packages_list:
            continue
        deps = pkg.get_dependencies()
```

```
        for dep in deps:
            if dep not in packages_to_check:
                packages_to_check.append( dep )
                dep_chain.append( dep.name )
        final_packages_list.append( pkg )
    for p in final_packages_list:
        print( p.name )
    return 0

if __name__=='__main__':
    main()
```

Sample Code Explanation

For those still struggling, here's a step by step breakdown of what this sample code is doing to illustrate the algorithm used for dependency resolution and circular dependency detection by DON.

1. First, we initialize two key lists:
    - final_packages_list[] holds packages in order they need installing
    - packages_to_check[] acts as our processing queue
    - dep_chain[] tracks packages in current dependency chain to detect cycles

2. We start with our requested package (httpd) and:
    - Put it in packages_to_check[] as our first package to process
    - Add its name to dep_chain[] to begin tracking dependencies

3. Then for each package in packages_to_check[]:
    - Check if package name exists in dep_chain[] (except last entry)
        - If found earlier in chain: we have a circular dependency – raise errorCheck

    - if package already in final_packages_list[]
        - If found: skip as we already handled it

4. If package passes those checks:
    - Get its dependencies via get_dependencies()
    - For each dependency:
        - Add to packages_to_check[] if not already there
        - Add its name to dep_chain[]
    - Add current package to final_packages_list[]
5. Result:
    - final_packages_list[] has packages in correct install order
    - Dependencies appear before packages that need them
    - Circular dependencies are caught during resolution

In this way, while it has a somewhat complex justification and capability, it is demonstrated simply.