Mandatory assignment 2 - inf2220 2008

Bjarne Holen

Deadline: Friday 7. November

This assignment should be solved individually, and handed to the teaching assistant responsible for your group.

Building a Package Management System

In this assignment we are going to build a package management system, which is a program that helps users (usually an administrator) install programs or resources. The word package will be used to denote either a program or some other component that programs can require to work properly, i.e., resources like libraries or fonts. The program we are going to construct in this assignment depends on a Java Virtual Machine, and the Java core library in order to work for instance. An installer (package management system) keeps track of the dependencies between packages and keeps your system free from broken packages. A broken package in this context is a program that will not work, since it requires a missing resource.

There are many package management systems around, the Linux installations at the University uses RPM (Red Hat Package Manager), type `rpm -qa` on the command line, and you will get a listing of the currently installed packages. On some operating systems programs typically come with an installer that contains its dependencies in case they are not present (Windows). Can you think of any problems which can occur with this approach?

Dependency graphs, like the one we are going to construct in this assignment, have a wide variety of applications; project planning, job scheduling, and is also an intricate part of software development (compilation, organizing imports…)

What is a Package?

In a real package management system, packages are typically some sort of compressed archive (tar, zip) containing the files of the package. And when a package is installed the archive is decompressed into the directory tree on the system. Type `rpm -qf /bin/ls` on the command line on one of the Linux machines at the University, and you will see what package this file belongs to. And you could type `rpm -ql coreutils` to see all the files belonging to this package, which was decompressed into the local file system when the package was installed.

Our packages will be a bit simpler and will only contain a list of dependencies, and a description.

```
# File xmlutils.pkg
#
depends: libxml
description: handy xml utilities
```

The fist two lines are considered comments since they start with a `#` sign, and are ignored. But all files in our repository will be files like this, which state their dependencies and description, both fields can be empty.

Note that for every dependency there is a reverse dependency, so not only do we have to make sure that the xml library `libxml` is installed prior to `xmlutils`, but we also have to make sure to remove `xmlutils` if `libxml` is removed.

To list dependencies for *emacs* with RPM: `rpm -q --requires emacs` and to list reverse dependencies type: `rpm -q --whatrequires emacs`

Existing Code

This assignment has quite a bit of existing code, which you will extend in order to create the installer. It's quite common that students dislike the idea of starting on these half-finished projects, but it does reduce the size of the task and it forces some structure on the project. This will in turn make it possible for your teaching assistants to read your source code faster and thus help you faster, and it is far more realistic then starting out with no source code.

The code that is handed out is able to do some of the work, but the dependency graph is missing, so any package can be installed and any package can be removed.

Ant

Since our project contains 14 classes spread out on different packages, compilation with *javac* is a bit impractical. *Ant* is a tool that helps you compile Java projects, similar to *make* for C and C++. It is also the default tool to build projects with most IDEs like Netbeans and Eclipse. Compiling source code with Ant is not hard, either with or without an IDE. Using Ant from the command line is shown in a video tutorial that can be found at `http://bjarneh.at.ifi.uio.no`

Dependency Graph

In our package management system each package can list its own dependencies, this information can be used to construct a dependency graph. Nodes in the dependency graph represents packages, and their dependencies are the nodes that can be reached by following the directed incoming edges. Below is a figure which hopefully illustrates how dependencies are interconnected, the dependencies are simplified, `rpm -q --requires eclipse-platform` to list the actual dependencies.

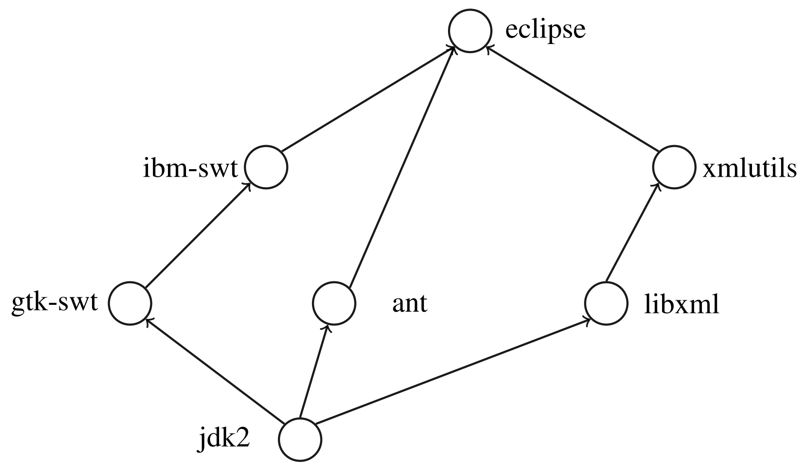

Figure 1: Illustration of dependencies

As you can see from the figure, Eclipse depends on packages which themselves depend on other packages. When our installer is asked to install Eclipse; we will have to traverse the incoming edges, and make sure everything in our *path* is installed. The sum of packages that can be reached this way, is what we call its dependencies, i.e., not only its direct dependencies but the recursive set of all elements which can be reached. Naturally a cycle inside our dependency graph would be critical, since a recursive traversal over a cycle would be never ending. We must ensure that no cycles are contained inside our dependency graph, this is not something we can take for granted.

To locate the reverse dependencies of a package, we follow the directed graph in the opposite direction (in the arrow direction). As you can see, all the packages in the figure on the previous page will be broken if we remove the Java Development Kit (jdk2) package. This is something we want to avoid, so we have to calculate the set of reverse dependencies before packages are removed.

Representation

So how do we represent a graph like this, and how do we calculate the dependencies and reverse dependencies? Naturally there is more then one way to do it, but from the task of locating dependencies and reverse dependencies we should focus on a structure which is easily traversable in both directions. Another thing worth noticing is that package names are unique, which means that we can use a `HashMap<String, Node>` to store pointers to all our nodes for quick retrieval. Once we have gotten a hold of the desired node, we can gather information about dependencies by simply flipping all the edges in Figure 1 to produce a structure like the one shown in Figure 2.

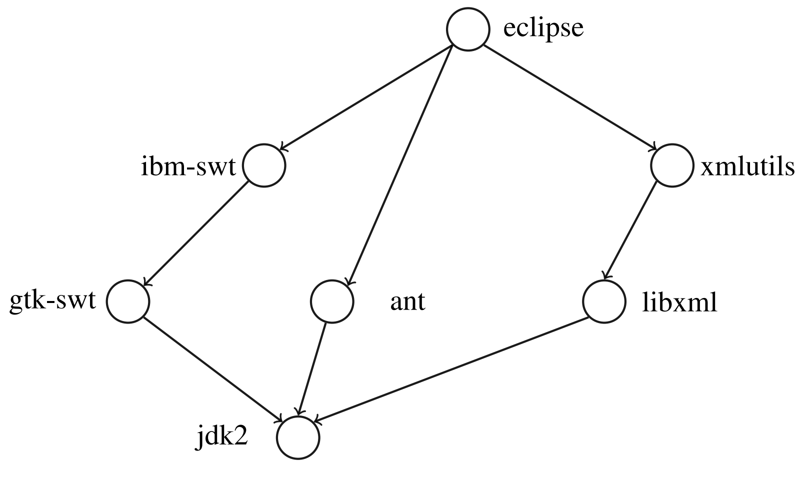

Figure 2: Representation of dependency graph

By stating that we can reverse all the edges in our directed acyclic graph, we also claim that this reversal will not produce cycles, can you prove this property?

A recursive traversal starting at the top node of Figure 2 (*eclipse*), will produce the set of elements which eclipse depends on. In other words, we reverse all the edges in our original dependency graph (Figure 1) in order to get a data structure which is able to generate the set of dependencies we are looking for. Note the use of the word *set* here, it can be taken literally. We can use a *set* in order to avoid adding the same element over and over. A recursive traversal of all paths (starting from the top) would add the last node three times, since all paths lead to the jdk2 package. But if we insert our elements into an instance of a class which implements the `java.util.Set` interface, we can eliminate multiple occurrences.

Where does this leave us with the reverse dependencies? This is basically the same situation, we have to traverse the (reverse) dependency graph in the opposite direction in order to gather the information we are looking for. The actual reverse dependency graph is Figure 2, but we have to flip all the edges here as well, so we end up with our original dependency graph, i.e., a structure like the one we have in Figure 1.

Naturally we need to know something about the dependencies for us to be able to construct the edges of the graph, but this information is given to us by the PackageManager, note that every dependency generates a reverse dependency.

One could argue that our graph is no longer a directed graph since we have pointers in both directions, but we make a distinction between parents and children, which means that loops will not occur with this approach. This is why the graph class got the name `DoubleDAG` since it is in some sense two directed acyclic graphs (DAG) using the same nodes.

Getting Started

The first thing you need is the source code, which can be found here:
(you should be able to unpack at least one of these)

- `http://bjarneh.at.ifi.uio.no/oblig2.zip`
- `http://bjarneh.at.ifi.uio.no/oblig2.tgz`

These files are compressed archives (zip,tar) which contain an Ant script (build.xml), the source code and an explanation of how to get started (README.txt).

`tar -xvzf oblig2.tgz` to unpack tar archive

`unzip oblig2.zip` to unpack zip archive

Source code is organized in a standard Java manner, where package names and name-spaces follow the directory structure and the URL of our institution backwards (ifi.uio.no becomes no.uio.ifi). Try to **grep** after TODO inside the source tree, this will hopefully get you going either you are using and IDE (Netbeans, Eclipse, IntelliJ, JBuilder) or a text editor (Emacs, Vim, TextPad, Gedit).

General Information

It is important to note that this assignment is given for the first time, which means that there will probably be some issues that we did not foresee. It also means that your feedback is highly encouraged and needed in order for this assignment to improve. And finally it means that the skill level needed to solve this assignment has not been tested extensively. In other words, it may be a bit difficult, or it may be too simple. A couple of optional assignments will be given in case you find this task to easy.

A Small Note on Efficiency

Hopefully this assignment highlights some aspects of efficiency which we have not dealt with during this course. We mainly deal with algorithms specially designed to optimize a specific operation this semester, but there are also some design patterns which will help you write more efficient code. In this assignment there are at least two such patterns.

1. Don't calculate anything until it is absolutely necessary
2. Cache calculations which are static

As an example of 1) we do not calculate the dependency graph until we actually need it. Maybe the user just wants to perform a search to look for a package, or maybe he just wants to list what packages he already has installed. In either case, the dependency graph is not needed.

As an example of 2) the content of the repository can be considered static for a session, i.e., we can assume that the content of the repositories stays the same for the duration of our program execution. So once we have read the content of the repository and calculated our dependency graph, we cache it (store it), i.e., we do not rebuild this structure during our session, unless the user asks for an update.

Topological Sort

A topological sort is an ordering of all the elements in in a directed acyclic graph such that no element is traversed before all its predecessors or dependencies are traversed. Note that there can be more than one such ordering of elements, here are a few examples from Figure 1.

- jdk - ant - libxml - xmlutils- gtk-swi - ibm-swi - eclipse
- jdk - gtk-swi - ibm-swi - libxml - xmlutils - ant - eclipse
- jdk - gtk-swi - ibm-swi - ant - libxml - xmlutils - eclipse

Where does this fit into the picture? This is used by installers in this situation: We have located a set of missing dependencies, and we need to install package A, B and C to fulfill requirements of package D, but in which order should we install A, B and C? Or with an example from Figure 1, we are missing `ant` and `xmlutils` in order to install `eclipse`, which of these should be installed first? A topological sort preserves the ordering of the nodes, i.e., for any $A < B$ in the dependency graph, we know that A will come before B in the topological sort. If you look at the different orderings listed you will see that `ant` proceeds `xmlutils` in the first ordering, but not in the second. What does that mean? It means that these two nodes are unrelated, so they can be installed in any order. The topological ordering of the elements gives us certainty that we do not violate any such dependency requirements between nodes.

Optional Assignments

1. Add a topological sort to your dependency graph
2. Add a graphical user interface
3. Modify `GetOpt` to handle multiple short-options concatenated

If anyone produces a really good solution for one of the last two optional assignments, your solution will be included into this project the next time it is given, with your name on it.

2. Use the Java Swing libraries to create a platform independent user interface. You are pretty much free to do this as you please, and this should be the default interface the user enters if no command line arguments are given.

3. `GetOpt` is the option parser in this assignment, and as you may know it is common that one can concatenate several short options. As an example, one can write `ls -l -a` in a short form as `ls -la` or one can write `tar -x -v -z -f` as `tar -xvzf`. This could be done with some preprocessing of the input arguments, before we start parsing them inside `GetOpt.parseArgv` for instance.

Handing in Your Assignment

Before you hand in you code, you must document your work. This should be done in a standard Javadoc fashion. If your teaching assistant should be aware of anything in particular concerning your project, you can add a small description inside the README.txt file which is located in the root directory.

When you finish your project, you should make an archive (either zip or tgz) with your solution and email it to your teaching assistant. You should run the Ant **clean** target before you create this archive, since your teaching assistant will be able to construct all the compiled code and documentation on his own.

Remember to take **backups** of your progress as you implement your solution, this is especially important if you don't use the machines here at the University for your work.

So a typical procedure before you hand in your assignment, could be something like this, assuming that your username is **olanormann** and that your assignment is still inside the directory called oblig2.

```
> cp -a oblig2  olanormann
> tar -cvzf olanormann.tgz olanormann
```

Now it should be a matter of sending the file olanormann.tgz to your teaching assistant. If something goes wrong here, your original source will still be in place in the oblig2 directory.

Help

For this assignment an IRC channel has been created, it was actually created for similar reasons when I helped out with another course, so it bares the name `#logikk`. So by connecting to `irc.ifi.uio.no` and joining this channel you should be able to get in touch with someone which is able to help you. Perhaps you can help each other out as well. If nobody answers your question right away, then you should check out this link:

`http://bjarneh.at.ifi.uio.no/other/utils/logger.php`

There you will find a logger for this channel, so everything written into the public domain of this channel is logged to this website. This should also be the first place you look if you have a question, since it may already be answered.

Good luck.

`bjarneh`