ABout Time Linux dependency engine requirements (v0.2) Groep 1

Mark Adriaanse markadriaanse@student.ru.nl
Marco Blommert marco@mbmidway.demon.nl
Evelien Roos me@evelienroos.nl
Eric Schabell ericS@abtlinux.org
Bernard van der Velden b.vandervelden@student.ru.nl

December 15, 2005

Contents

1 Introduction **4**

2 Problem Statement **4**

3 Stakeholder Analysis **5**

4 Mission, Vision and Values **6**
4.1 Mission 6
4.2 Vision 6
4.3 Values 6

5 Statement of Work **7**
5.1 Scope 7
5.2 Objectives 7
5.3 Application Overview 7
5.4 User demography 7
5.5 Constraints 7
5.6 Assumptions 7
5.7 Staffing and cost 8
5.8 Deliverable Outlines 8
5.9 Duration 8

6 Risk Analysis **9**
6.1 Language 9
6.2 LaTeX 9
6.3 Sickness 9
6.4 Bad Communication 9
6.5 Busy 9
6.6 Domain knowledge 10
6.7 Electronic conversation 10

7 Use Case Survey **11**
7.1 Request list of packages lacking dependencies 11
7.2 Request dependency-tree 11
7.3 Request dependency-tree uptree 12
7.4 Request dependency-tree downtree 12
7.5 Request number of pkg's uptree dependencies 13
7.6 Request number of pkg's downtree dependencies 13
7.7 Request buildorder 14
7.8 Select level of logging 14

8 Use Case Diagram **15**

9 Use Cases **16**
9.1 Request list of packages lacking dependencies 16
9.2 Request dependency-tree 17
9.3 Request dependency-tree uptree 18
9.4 Request dependency-tree downtree 20
9.5 Request number of pkg's uptree dependencies 22

9.6 Request number of pkg's downtree dependencies 23
9.7 Request buildorder 25
9.8 Select level of logging 27

10 Scenarios **28**
10.1 Request list of packages lacking dependency's 28
10.2 Request dependency-tree 28
10.3 Request dependency-tree uptree 29
10.4 Request dependency-tree downtree 31
10.5 Request number of pkg's uptree dependencies 32
10.6 Request number of pkg's downtree dependencies 34
10.7 Request buildorder 35
10.8 Select level of logging 38

11 Domain Models **39**
11.1 Request list of packages lacking dependencies 39
11.2 Request dependency-tree 39
11.3 Request dependency-tree uptree 40
11.4 Request dependency-tree downtree 40
11.5 Request number of pkg's uptree dependencies 41
11.6 Request number of pkgs downtree dependencies 41
11.7 Request buildorder 42
11.8 Select level of logging 42

12 Business Rules **43**

13 Non-functional Requirements **43**

14 Terminological Definitions **44**

1 Introduction

ABout Time Linux [AbT-Linux] is a fairly new Linux distribution started several years ago by Eric Schabell. After having worked for several years on other Linux distribution, Eric wanted to make a distribution of his own. A distribution that would be correctly documented and designed.
Like Eric most members of this project have worked on projects of other Linux distributions and have grown tired of working on badly documented designs. This leads to badly organized growth paths for these distributions and hard to manage code bases for the tools. So AbT-Linux has been designed from scratch following strict guidelines for requirements, design and coding. AbT-Linux will be a Linux distribution the way Eric thinks a Linux distribution should be.
One of the requirements for AbT-Linux, is a firm software package management system [abt package manager]. This is a package manager which is able to support users in using and managing the software packages.
The dependency-engine [depEngine] is a part of the abt package manager. The depEngine has to look for dependencies between software packages and must be able to create an ordered buildlist for changing software configurations on a AbT-Linux system. The depEngine will primary be used by the abt package manager.
The primary goal of this project is a well documented design, so others can create the depEngine as part of the easily maintainable software package management system. This in turn will be the foundation of AbT-Linux.

2 Problem Statement

The abt package manager is software to be developed for the managing of large amounts of software. It needs to have a dependency engine to create an overview of the packages (with their dependencies)in the system. The package manager needs this because it wants to know which packages can be influenced by a certain action he takes. The package manager wants to know what packages depend or rely on another package. The dependency engine (depEngine) is able to to generate this overview, which is a dependency tree for given packages. The dependency engine is an important part of the abt package manager. When users have large amounts of software on their computers it is hard to keep track of it during all changes and updates in these softwarepackages. Some packages are dependent on one another, the depEngine is an organizer of these.

3 Stakeholder Analysis

The stakeholders are everyone that can benefit from the depEngine. This benefit can be at different levels. By using the product or participating in the development for example.

Open Source community

The Open Source Community can profit of a firm software package management system. In current linux-distributions users often have to do this by hand. They have to manage their own dependencies when installing a new package. With the abt package manager the users don't have to concern about any dependencies and it will make updating and configuring of software on their computer easier.

Developers

The developers of the AbT-Linux distribution are also stakeholders. Not just by the development of the depEngine but by the result of it. If the depEngine is a great succes their names will be attached to it , but also if it is a complete failure. So the developers do have a certain stake in a well developed depEngine.

Future Developers

If in the future the depEngine should be modified the developers can benefit of if the product is well documented and designed. These developers will have benefit from the well written documentation. They can depend on a well documented and designed system which makes it easier to make additions or changes to the system. So the future developers do have a certain stake in a well developed depEngine.

Executive Sponsor

Another stakeholder is Eric Schabell who is the founder and executive sponsor of the AbT-Linux distribution. The depEngine is a primary tool to be used by the abt package manager. Eric Schabell wants the development of the depEngine to be well documented from the beginning of the design cycle. "We need to do as we promise, document and design a depEngine for a source-based Linux distribution."

4 Mission, Vision and Values

4.1 Mission

To make working with different Open Source applications easier, Eric Schabell started the developing of a software packages manager. An essential part of this package manager, abt package manager is the depEngine.

4.2 Vision

Eric Schabells vision is to develop a tool, called abt package manager which helps users to manage software packages. The abt package manager needs to have a depEngine to generate a dependency tree for given packages. This in order to organize the different dependencies between software and provide with a usefull buildorder to install, remove, rebuild, reconfigure, upgrade, downgrade and repair.

4.3 Values

It is important that the package manager is very easy to use. By making the package manager so easy more people will use it instead of trying to install, update and manage the packages by themselves. Also very important is that the program and the source-code will be distributed freely, one of the characteristics of Open Source software. The depEngine should provide a clear list of which program has a relation to another and what happens if one of them is altered. The package management tool will be an Open Source product for the Open Source Community. The goal of Open Source software to create a transparent and free software world. Both the source code as well as the program can be distributed freely and users can help to develop the design. Developers and requirements engineers should follow the KISS principle: Keep It Simple Silly. It is obligatory that all documentation should be written in Latex because the executive sponsor wants it.

5 Statement of Work

5.1 Scope

The scope of the project is the dependency engine (depEngine). This statement of Work (SOW) defines the effort required for the gathering and composing requirements for the depEngine. In this project the main object is the depEngine. Because the depEngine is a part of the abt package manager the interaction between these two is also a part of this project. Because the depEngine gathers information about the dependencies of the packages, the depenencies are regarded as a part of this project, the packages themselves are not. At the end of the project there will be set up a well founded set of requirements that can be handed over to the programmers so they can build the depEngine. The development of the source code not a part of this requirements engineering process.

5.2 Objectives

The objective of the project is to deliver a set of requirements for the depEngine. The objective of the depEngine is to provide a clear overview of different dependencies in order to support the abt package manager in fullfilling its task, which is the organizing of packages.

5.3 Application Overview

The abt package manager

At the end of the overall project the abt package manager will be ready to be launched into the Open Source community. A part of the abt package manager will be the dependency engine (depEngine). The abt package manager supports the user in organizing packages and their depedendencies.

The depEngine

The depEngine is a part of the abt package manager. The depEngine gives an insight of the packages with their dependencies.

5.4 User demography

The user of the depEngine is the abt package manager. The package manager can be used by people who download it from the Internet or who receive it as a part of the AbT-linux distro.

5.5 Constraints

The documentation should all be written in LaTeX. All documentation will be written in English. The most prefered way of communication between the requirements team and the executive sponsor is through email.

5.6 Assumptions

We assume that every group member will have an equal part in the requirements engineering process.

5.7 Staffing and cost

Since the students can earn their ECTS with writing a good RE-document this is their reward. Eric Schabell and his ABT project team all work on this project voluntarily.

5.8 Deliverable Outlines

During the development of the requirements document there will be several deliverables, the most important are the use cases and their scenarios. The development of the requirements will have an iterative and incremental character. During the different phases the document will grow but also be refined. Different iterations are: the facade phase (deadline 10-13-2005), filled phase (11-17-2005) and the focused phase (12-15-2005).

5.9 Duration

The requirements engineering for the depEngine ends on december 15th, when the deadline of the focused phase is reached.

6 Risk Analysis

The risk analysis is used to get an overview of the possible risks that can influence the development of the application.

6.1 Language

The fact that the documents have to be written in English can be a potential risk. Most of the students do speak English but to write reports in English is new to most of the students.
Days lost if it occurs: 2
Likelihood it will happen:10
Riskrating: 4

6.2 LaTeX

LaTeXis new to all of the students and is seen as a risk. No member of our group has used LaTeXbefore but we think we should be able to cope with LaTeX.
Days lost if it occurs: 4
Likelihood it will happen: 40
Risk rating: 16

6.3 Sickness

People get sick, especially in this period of the year. The other group members will try to deal and help with the tasks this person was ment to do.
Days lost if it occurs: 5
Likelihood it will happen: 25
Risk rating: 125

6.4 Bad Communication

Because the projectgroup mainly works at home and only see eachother once a week, it is very important that the communication between the groupmembers is well regulated.
Days lost if it occurs: 10
Likelihood it will happen: 60
Risk rating: 200

6.5 Busy

The project is one of the many projects and things for the developers to do. When the developers have several deadlines in the same period this can stress on this project. This means that deadlines broken or that deliverables are not finished.
Days lost if it occurs: 14
Likelihood it will happen: 40
Risk rating: 250

6.6 Domain knowledge

The knowledge of the domain in general is very poor. It is essential to have at least some knowledge about the domain.
Days lost if it occurs: 20
Likelihood it will happen: 80
Risk rating: 500

6.7 Electronic conversation

Because conversation with the founder and executive sponsor is mainly by e-mail and other electronic ways there are very few moments of real conversation where live questions can be asked. Often this works faster and better.
Days lost if it occurs: 3
Likelihood it will happen: 80
Risk rating: 50

7 Use Case Survey

7.1 Request list of packages lacking dependencies

Use case number	1
Use case name	Request list of packages lacking dependencies
Initiating actor	Package manager
Description	The package manager requests a list of all packages lacking dependencies. The depEngine returns a list of available packages lacking dependencies (orphans) to the package manager.
Completeness	Complete
Maturity	Mature
Caution	n/a
Dependency	n/a
Source	From Eric Schabells presentation
Comments	n/a

7.2 Request dependency-tree

Use case number	3
Use case name	Request dependency-tree
Initiating actor	Package manager
Description	The abt package manager requests a dependency tree of the depEngine. The depEngine generates a complete dependency-tree from the uptree and downtree dependency trees requested for each single installed package in the list. The depEgine returns this tree to the package manager. The tree is a tree-scheme with all the packages and the relations between them. Each package in the tree must contain information about the software-version in the package, on which packages it depends uptree and wich packages are depending on it downtree (this includes optional packages).
Completeness	Complete
Maturity	Mature
Caution	n/a
Dependency	This use case depends on use case 4 and 5
Source	From Eric Schabell's presentation
Comments	Versionnumbers are also regarded as dependencies. Different versionnumbers of a software-program are regarded as different packages, example: apache 1.3 and 2.0 are different packages.

7.3 Request dependency-tree uptree

Use case number	4
Use case name	Request dependency-tree uptree
Initiating actor	Package manager
Description	The depEngine generates a dependency-tree from uptree dependencies of all available packages. The depEgine returns this tree to the package manager. The tree is a tree-scheme with all the packages and the relations between them. Each package in the tree must contain information about the softwareversion in the package, on which packages it depends uptree (this includes optional packages)
Completeness	Complete
Maturity	Mature
Caution	n/a
Dependency	n/a
Source	From Eric Schabells presentation, abtlinux.org
Comments	Versionnumbers are also regarded as dependencies. Different versionnumbers of a software-program are regarded as different packages, example: apache 1.3 and 2.0 are different packages.

7.4 Request dependency-tree downtree

Use case number	5
Use case name	Request dependency-tree downtree
Initiating actor	Package manager
Description	The abt package manager requests a dependency tree downtree of a certain package from the depEngine. The depEngine generates a dependency-tree from downtree dependencies of all available packages for this certain package. The depEgine returns this tree to the package manager. The tree is a tree-scheme with all the packages and the relations between them. Each package in the tree must contain information about the softwareversion of the package, on which packages it depends downtree (this includes optional packages).
Completeness	Complete
Maturity	Mature
Caution	n/a
Dependency	n/a
Source	From Eric Schabells presentation, abtlinux.org
Comments	Versionnumbers are also regarded as dependencies. Different versions of a software-program are regarded as different packages, example: foo 1.3 and 2.0 are two different packages.

7.5 Request number of pkg's uptree dependencies

Use case number	6
Use case name	Request number of pkg's uptree dependencies
Initiating actor	Package manager
Description	The package manager requests a number of packages depending uptree on a package. The depEngine counts and returns the number of packages that are dependent uptree of the given package.
Completeness	Complete
Maturity	Mature
Caution	n/a
Dependency	The package manager needs to specify the package.
Source	From Eric Schabells presentation
Comments	n/a

7.6 Request number of pkg's downtree dependencies

Use case number	7
Use case name	Request number of pkg's downtree dependencies
Initiating actor	Package manager
Description	The package manager requests a number of packages depending downtree on a single package. The depEngine must return the number of dependencies downtree.
Completeness	Complete
Maturity	Mature
Caution	n/a
Dependency	The package manager needs to specify which package.
Source	From Eric Schabells presentation
Comments	n/a

7.7 Request buildorder

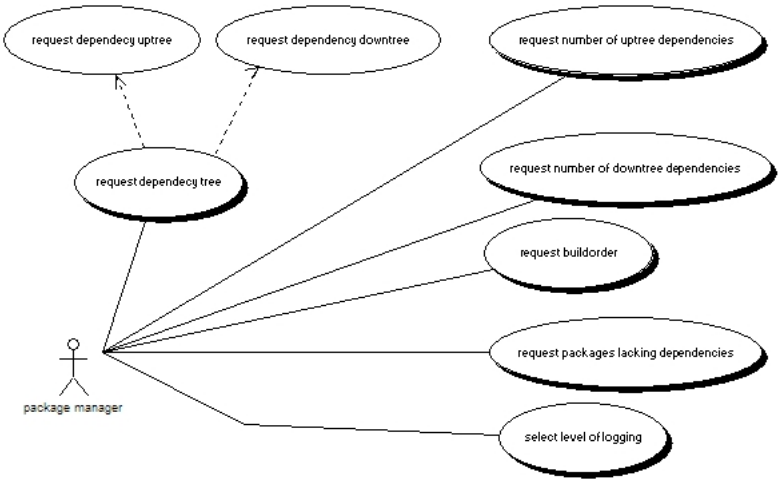
Use case number	8
Use case name	Request buildorder
Initiating actor	Package manager
Description	The package manager requests a sorted list for the dependencies of a package. The depEngine delivers this buildorder. The buildorder illustrates the order of the packages (depending on their dependencies and versions) that the package managers want to build. The depEngine has to search the whole dependency tree and find the dependencies and versions of each package, before the depEngine gives the right order. The right order is the order in which the least possible packages have to be build repeatedly.
Completeness	Complete
Maturity	Mature
Caution	n/a
Dependency	n/a
Source	From the forum on abtlinux.org
Comments	This will be an list of the order of the packages.

7.8 Select level of logging

Use case number	9
Use case name	Select level of logging
Initiating actor	Package manager
Description	The package manager selects the level of detail for generating logs while using the dependency-engine. These logs can be used for tracking errors.
Completeness	Complete
Maturity	Mature
Caution	n/a
Dependency	n/a
Source	From Eric Schabells presentation
Comments	n/a

8 Use Case Diagram

Below the Use Case Diagram:



9 Use Cases

9.1 Request list of packages lacking dependencies

Use Case name:	Request list of packages lacking dependencies
Iteration:	Focused
Summary:	The package manager requests a list of all packages lacking dependencies. The depEngine returns a list of available packages lacking dependencies (orphans) to the package manager.
Basic course of events:	1. The package manager requests a list of available packages lacking dependencies from the depEngine. 2. The depEngine loads the list of all packages. 3. The depEngine searches for packages without any dependencies to other packages. 4. The depEngine generates a list of these packages. 5. The depEngine returns this list to the package manager.
Alternative paths:	1. The depEngine can't find a list of all packages (step 2), the depEngine will report to the package manager it can't find any packages. 2. The depEngine can't find packages without dependencies (step 3), the depEngine reports to the package manager that there are no orphaned packages (packages lacking dependencies).
Exception paths:	n/a
Extention points:	n/a
Triggers:	The package manager wants a list of packages lacking dependencies.
Assumptions:	All actions (step 1 though 3 in BCoE) taken in this use case will be logged.
Preconditions:	n/a
Postconditions:	The package manager retrieves a list of available packages lacking dependencies.
Related business rules:	n/a
Author:	Groep 1
Date:	29 November 2005

9.2 Request dependency-tree

Use Case name:	Request dependency-tree
Iteration:	Focused
Summary:	The abt package manager requests a dependency tree of the depEngine. The depEngine generates a complete dependency-tree from the uptree and downtree dependency trees requested for each single installed package in the list. The depEgine returns this tree to the package manager. The tree is a tree-scheme with all the packages and the relations between them. Each package in the tree must contain information about the software-version in the package, on which packages it depends uptree and wich packages are depending on it downtree (this includes optional packages).
Basic course of events:	1. The package manager submits the package from which the dependency tree (up and down) is required. 2. The depEngine loads the dependency uptree for the given (input) package. 3. The depEngine loads the dependency downtree for the given (input) package. 4. The depEngine puts both trees (up and down) together. The point of connection will be the input package itself. 5. The depEngine returns this dependency tree to the package manager.
Alternative paths:	n/a
Exception paths:	1. If the depEngine fails in loading the up- or downtree dependency-tree of a certain package in the list, an error is reported to the package manager.
Extension Points:	n/a
Triggers:	• The package manager requests a dependency-tree.
Assumptions:	All actions taken in this use case will be logged.
Preconditions:	A downtree and uptree dependency tree must be available.
Post conditions:	The package manager receives the complete dependency tree of the input package.

Related business rules:	n/a
Author:	Groep 1
Date:	14 December 2005

9.3 Request dependency-tree uptree

Use Case name:	Request dependency-tree uptree
Iteration:	Focused
Summary:	The depEngine generates a dependency-tree from uptree dependencies of all available packages. The depEgine returns this tree to the package manager. The tree is a tree-scheme with all the packages and the relations between them. Each package in the tree must contain information about the softwareversion in the package, on wich packages it depends uptree (this includes optional packages)
Basic course of events:	1. The package manager requests the dependency-tree uptree. 2. The depEngine searches the higher levels for the packages relying on each other. 3. The depEngine lists these packages and their dependency. 4. The depEngine checks for cycled dependencies and lists these packages and their dependency. 5. The depEngine generates the dependency-tree uptree. 6. The depEngine remembers the tree for later usage. 7. The depEngine returns the dependency-tree to the package manager.

Alternative paths:	n/a
Exception paths:	1. The depEngine searches the higher levels for the packages relying on each other (BCoE: 2). In case the depEngine finds out a package is missing files, it reports an error to the package manager. 2. The depEngine checks for cycled dependencies and lists these packages and their dependency (BCoE: 4). The depEngine reports an error to the package manager.
Extension Points:	n/a
Triggers:	• The package manager requests the dependency-tree uptree.
Assumptions:	All actions taken in this use case will be logged.
Preconditions:	A valid package-list must be available.
Post conditions:	The package manager recieves a dependency uptree.
Related business rules:	n/a
Author:	Groep 1
Date:	15 December 2005

9.4 Request dependency-tree downtree

Use Case name:	Request dependency-tree downtree
Iteration:	Focused
Summary:	The abt package manager requests a dependency tree downtree of a certain package from the depEngine. The depEngine generates a dependency-tree from downtree dependencies of all available packages for this certain package. The depEgine returns this tree to the package manager. The tree is a tree-scheme with all the packages and the relations between them. Each package in the tree must contain information about the softwareversion of the package, on which packages it depends downtree (this includes optional packages).
Basic course of events:	1. The package manager requests the dependency-tree downtree of a package. 2. The depEngine loads the package list. 3. The depEngine searches the lower levels for the packages relying on each other. 4. The depEngine lists these packages and their dependency. 5. The depEngine checks for cycled dependencies and lists these packages and their dependency. 6. The depEngine generates the dependency-tree downtree. 7. The depEngine remembers the tree for later usage. 8. The depEngine returns the dependency-tree to the package manager.

Alternative paths:	n/a
Exception paths:	1. The depEngine searches the lower levels for the packages relying on each other (BCoE: 2). In case the depEngine finds out a package is missing files, it reports an error to the package manager. 2. The depEngine checks for cycled dependencies and lists these packages and their dependency (BCoE: 4). The depEngine reports an error to the package manager.
Extension Points:	n/a
Triggers:	• The package manager requests the dependency-tree downtree
Assumptions:	All actions taken in this use case will be logged.
Preconditions:	A valid package-list must be available.
Post conditions:	The package manager receives a dependency tree downtree.
Related business rules:	n/a
Author:	Groep 1
Date:	29 November 2005

9.5 Request number of pkg's uptree dependencies

Use Case name:	Request number of pkg's uptree dependencies.
Iteration:	Focused.
Summary:	The package manager requests a number of packages depending uptree on a single package. The depEngine must return the number of dependencies uptree.
Basic course of events:	1. The package manager submits the name of the package of wich the number of uptree dependencies must be counted. 2. The depEngine identifies the position of the submitted packagename in the dependency-tree. 3. The depEngine counts the number of packages uptree which have a dependency with the given package. 4. The depEngine returns the number of counted dependencies to the package manager.
Alternative paths:	1. When there is no packagename submitted, the depEngine must tell de package manager to select a package for counting dependencies. 2. When there is no dependency-tree uptree, request the dependency-tree uptree with the usecase "Request dependency-tree uptree".
Exception paths:	1. When counting of the dependencies somehow fails, return an error to the package manager, log this event and stop the usecase.
Extension Points:	n/a
Triggers:	• The package manager wants to know the number of uptree dependencies of a package.
Assumptions:	All actions (step 1 though 4 in BCoE) taken in this use case will be logged.
Preconditions:	A package for counting the uptree dependencies of must be given to the depEngine.

Post conditions:	A number of counted dependencies uptree of a package is given to the package manager.
Related business rules:	n/a
Author:	Groep 1
Date:	29 November 2005

9.6 Request number of pkg's downtree dependencies

Use Case name:	Request number of pkg's downtree dependencies.
Iteration:	Focused.
Summary:	The package manager requests a number of packages depending downtree on a single package. The depEngine must return the number of dependencies downtree.
Basic course of events:	1. The package manager submits the package from which the number of downtree dependencies must be counted. 2. The depEngine reads the package description of the submitted package, to pick up the dependencies down tree. 3. The depEngine counts the number of dependencies downtree from the submitted package. 4. The depEngine returns the number of counted dependencies to the package manager.
Alternative paths:	1. When there is no package submitted, the depEngine must tell the package manager to select a package for counting dependencies.
Exception paths:	1. When counting the dependencies somehow fails, return an error to the package manager, log this event and stop the usecase.
Extension Points:	n/a
Triggers:	• The package manager wants to know the number of downtree dependencies of a package.

Assumptions:	All actions (step 1 though 4 in BCoE) taken in this use case will be logged.
Preconditions:	A package for counting the downtree dependencies of must be given to the depEngine.
Post conditions:	A number of counted dependencies downtree of a package is given to the package manager.
Related business rules:	n/a
Author:	Groep 1
Date:	14 December 2005

9.7 Request buildorder

Use Case name:	Request buildorder
Iteration:	Focused.
Summary:	The package manager requests a buildorder from the depEngine. The depEngine must return a buildorder to the package manager that follows the dependency-tree from the top to the bottom, without breaking dependencies. The list shows in which order the packages have to be built without breaking the dependencies. The depEngine has to search the whole dependency tree and find the dependencies and versions of each package. The right order to rebuild is given by the depEngine.
Basic course of events:	1. The depEngine receives a list from the abt package manager (foo, foo1, foo2) to be sorted for building 2. The depEngine searches the dependency up-tree and the dependency down-tree for each package. 3. If there are no dependencies for these packages the buildorder is the same as the list given by the package manager. 4. When the depEngine finds a dependency for 'foo', the depEngine lists this dependency. 5. The depEngine looks at the dependency to see what kind of dependency it is and on which version the dependency is based on. 6. When the depEngine searches the tree further and finds another dependency for that package, the depEngine looks at what kind of dependency this is. 7. The depEngine analyses the dependency and puts the packages in the right builorder. 8. The depEngine checks if the versions are correct and generates a buildorder for the given packages. 9. When the depEngine has finished searching the tree, it returns the list to the package manager.

Alternative paths:	n/a
Exception paths:	1. When the depEngine receives an incorrect value from the dependency-tree, the use-case must be stopped and an error must be returned to the package manager. This is important because an incorrect buildorder can cause unwanted orders or repeated building of packages.
Extension Points:	n/a
Triggers:	• The package manager wants a buildorder for packages that have to be built.
Assumptions:	• The version of the package is registered by the depEngine.
Preconditions:	• A list of packages is given by the package manager.
Post conditions:	The package manager receives a sorted buildlist.
Related business rules:	The depEngine registers the versions of the packages.
Author:	Groep 1
Date:	15 December 2005

9.8 Select level of logging

Use Case name:	Select level of logging
Iteration:	Focused
Summary:	The package manager selects the level of detail for generating logs while using the dependency-engine. These logs can be used for tracking errors.
Basic course of events:	1. The package manager submits a logging-level to the depEngine. 2. Every time the depEngine processes an action, this action must be logged to a logfile.
Alternative paths:	1. The package manager decides the actions don't have to be logged. 2. When the the logfile can not be written, an error must be given to the package manager.
Exception paths:	n/a
Extension Points:	n/a
Triggers:	• The package manager wants to change the logging-level.
Assumptions:	• The destination of the logfile is given by the package manager.
Preconditions:	A writable logfile must be available.
Post conditions:	The logging-level of the depEngine will be changed to a given logging-level.
Related business rules:	The logs must be stored in a standard logging-format.
Author:	Groep 1
Date:	27 November 2005

10 Scenarios

10.1 Request list of packages lacking dependency's

This scenario follows the basic course of events. The depEngine returns a list of available packages lacking dependencies to the package manager.

1. The abt package manager requests a list of available packages without dependencies.
2. The depEngine tries to load the package-list containing all packages.
3. The depEngine searches the list for packages that have no dependencies, this could be ″foo1″ and ″foo2″.
4. The depEngine returns the list containing ″foo1″ and ″foo2″.
5. The depEngine returns this packagelist to the package manager.

This scenario will follow alternative path 1. The depEngine tries to return a list of available packages lacking dependencies to the package manager.

1. The abt package manager requests a list of available packages without dependencies.
2. The depEngine tries to load the package list containing all packages.
3. The depEngine can't find a packagelist.
4. The depEngine returns a message to the package manager containing the error it can't find any packages.

This scenario will follow alternative path 2. The depEngine tries to return a list of available packages without dependency's to the package manager.

1. The abt package manager requests a list of available packages without dependencies.
2. The depEngine tries to load the package list containing all packages.
3. The depEngine searches the list for packages that have no dependencies, none are found.
4. The depEngine returns a message to the package manager containing the message it can't find any packages lacking dependencies.

10.2 Request dependency-tree

This scenario follows the basic course of events. The depEngine generates a complete dependency-tree from the (optional) downtree and (optional) uptree dependencies. The package manager needs the dependency-tree to see which packages have dependency's with eachother and what type the dependency is. The depEgine returns this tree to the package manager.

1. The abt package manager submits the package 'foo' to the depEngine for requesting the complete up and down tree of the given package 'foo'.
2. The depEngine successfully loads the already existing uptree for the package 'foo'.
3. The depEngine successfully loads the already existing downtree for the package 'foo'.
4. The depEngine puts both tree together. The point where both trees will be connected to each other is 'foo'.
5. The depEngine returns the combined tree to the package manager.

This scenario follows the exception path. The depEngine can not create a complete dependency-tree, because it fails loading the uptree.

1. The abt package manager submits the package 'foo' to the depEngine for requesting the complete up and down tree of the given package 'foo'.
2. The depEngine fails to load the uptree for the package 'foo'. There can be many reasons to why this fails. It might be the case there simply is no uptree available of 'foo'. Another possibility is that the uptree data is corrupted, or it might be the case that there is an interruption in the loading process. Despite the precondition of this usecase, these still are possible scenarios.
3. The depEngine will not load the already existing downtree for the package 'foo', because it can not create a complete dependency tree anyway.
4. The depEngine can not put both trees together.
5. The depEngine returns the result by giving an error to the abt package manager.

10.3 Request dependency-tree uptree

This scenario follows the basic course of events. The package manager needs the dependency-tree uptree to see which packages have dependencies with other packages, in levels above the level of the package and what type the dependencies are.

1. The AbT-Linux package manager requests the dependency-tree uptree.
2. The depEngine loads the package-list (with possibly changed packages).
3. The depEngine looks at the dependency info of a package in the packages in the above levels in the tree for the packages relying on each other.
4. The depEngine will find the dependency 'foo RO foo1' and sets the rebuild-property for package foo to true when foo1 has an altered configuration for example.

5. The depEngine searches the tree further up. The depEngine for instance processes 'foo DO foo1' and sets the rebuild-property for package foo to true because foo1 is upgraded.
6. The depEngine checks for broken dependencies. This occurs when a package has a dependency to a missing package, a package is missing files or a package failed an integrity-check.
7. When a broken dependency is found, the depEngine reports this to the package manager.
8. The depEngine checks for cycled dependencies. This situation occurs when a package foo depends on package foo1, and vice versa.
9. When a cycled dependency is found, the depEngine reports this to the package manager.
10. The depEngine generates the downtree-scheme. This scheme starts at the top with the packages that have the most packages relying on it. The packages with a larger number of DO dependencies are lower in the tree.
11. The depEngine stores the downtree in for later usage.
12. The depEngine returns the content downtree to the package manager.

This scenario follows exceptionpath 1.

1. The AbT-Linux package manager requests the dependency-tree uptree.
2. The depEngine want's to load the package-list (with possibly changed packages), but package "foo" is missing files.
3. The depEngine returns an error to the package manager that a file of package "foo" is missing.

This scenario follows exceptionpath 2.

1. The AbT-Linux package manager requests the dependency-tree uptree.
2. The depEngine loads the package-list (with possibly changed packages).
3. The depEngine looks at the dependency info of a package in the packages in the above levels in the tree for the packages relying on each other.
4. The depEngine will find the dependency "foo RO bar" and "bar RO foo", this is an cycled dependency.
5. The depEngine reports an error to the package manager telling it packages "foo" and "bar" have cycled dependencies.

10.4 Request dependency-tree downtree

This scenario follows the basic course of events. The package manager needs the dependency-tree downtree to see which packages have dependencies with other packages, in levels below the level of the package and what type the dependencies are.

1. The AbT-Linux package manager requests the dependency-tree downtree of a package 'foo1'.
2. The depEngine loads the package-list (with possibly changed packages).
3. The depEngine looks at the dependency info of a package in the packages in the lower levels in the tree for the packages relying on 'foo1'. The depEngine will find the dependency 'foo RO foo1' and sets the rebuild-property for package foo to true when foo1 has an altered configuration for example. The depEngine searches the tree further down. The depEngine for instance processes 'foo DO foo1' and sets the rebuild-property for package foo to true because foo1 is upgraded.
4. The depEngine checks for cycled dependencies. This situation occurs when a package foo depends on package foo1, and vice versa.
5. When a cycled dependency is found, the depEngine reports this to the package manager.
6. The depEngine generates the downtree-scheme. This scheme starts at the top with the packages that have the most packages with a relation with 'foo1'. The packages with a larger number of DO dependencies are lower in the tree.
7. The depEngine stores the downtree in for later usage.
8. The depEngine returns the downtree to the package manager.

This scenario follows the exception path 1. The depEngine searches the lower levels for the packages relying on each other (BCoE: 4). In case the depEngine finds out a package is missing files, it reports an error to the package manager.

1. The AbT-Linux package manager requests the dependency-tree downtree of a package 'foo1'.
2. The depEngine loads the dependency tree.
3. The depEngine looks at the dependency info of a package in the packages in the lower levels in the tree for the packages relying on 'foo1'. The depEngine will find the dependency 'foo RO foo1' and sets the rebuild-property for package foo to true when foo1 has an altered configuration for example. The depEngine searches the tree further down. The depEngine for instance processes 'foo DO foo1' and sets the rebuild-property for package foo to true because foo1 is upgraded.
4. The depEngine checks for cycled dependencies. This situation occurs when a package foo depends on package foo1, and vice versa.

5. The depEngine finds the dependency 'foo1 RO foo2' and sees 'foo2' is missing files.

6. The depEngine returns an error to the packagemanager.

This scenario follows the exception path 2. The depEngine checks for cycled dependencies and lists these packages and their dependency (BCoE: 5). The depEngine reports an error to the package manager.

1. The AbT-Linux package manager requests the dependency-tree downtree of a package 'foo1'.

2. The depEngine loads the dependency tree.

3. The depEngine looks at the dependency info of a package in the packages in the lower levels in the tree for the packages relying on 'foo1'. The depEngine will find the dependency 'foo RO foo1' and sets the rebuild-property for package foo to true when foo1 has an altered configuration for example. The depEngine searches the tree further down. The depEngine for instance processes 'foo DO foo1' and sets the rebuild-property for package foo to true because foo1 is upgraded.

4. The depEngine checks for cycled dependencies. This situation occurs when a package foo depends on package foo1, and vice versa.

5. The depEngine finds the dependency 'foo1 RO foo2' and finds the dependendcy 'foo2 RO foo1'.

6. The depEngine returns this cycled dependency to the packagemanager.

10.5 Request number of pkg's uptree dependencies

This scenario will follow the basic course of events. The package manager needs to know the number of uptree dependencies of the package 'foo' to know the influence editing the package could have.

1. The package manager requests the number of pkg's uptree dependencies and submits the name of the package 'foo' to the depEngine.

2. The depEngine looks for 'foo1' in the dependency-tree.

3. The depEngine searches uptree for packages having dependencies with 'foo1'.

4. If 'foo1 DO foo2' the number of package dependencies is 1.

5. If 'foo2 DO foo3' the number of package dependencies of 'foo' is 2.

6. If 'foo3 RO foo4' the number of package dependencies of 'foo' is 3.

7. This number is returned to the package manager.

8. The package manager knows that editing 'foo' will have influence on 3 dependencies uptree.

This scenario will follow alternative path 1. There is no packagename given to the depEngine.

1. The package manager requests the number of pkg's uptree dependencies but submits no packagename.
2. The depEngine asks the package manager for a package name.
3. The package manager submits a package name.
4. This use case is continued with the Basic Course of Events at step 1.

This scenario will follow alternative path 2. When there is no dependency-tree uptree, request the dependency-tree uptree with the usecase "Request dependency-tree uptree".

1. The package manager requests the number of pkg's uptree dependencies of the package 'foo'.
2. The depEngine looks at the dependency tree, if there is no dependency tree available, the depEngine starts to generate a dependency tree uptree.
3. The depEngine continues the Basic Course of Events at step 2.

This scenario follows the exception path 1. The depEngine searches the higher levels for the packages relying on each other (BCoE: 4). In case the depEngine finds out a package is missing files, it reports an error to the package manager.

1. The AbT-Linux package manager requests the dependency-tree uptree of a package 'foo1'.
2. The depEngine loads the dependency tree.
3. The depEngine looks at the dependency info of a package in the packages in the lower levels in the tree for the packages relying on 'foo1'. The depEngine will find the dependency 'foo RO foo1' and sets the rebuild-property for package foo to true when foo1 has an altered configuration for example. The depEngine searches the tree further down. The depEngine for instance processes 'foo DO foo1' and sets the rebuild-property for package foo to true because foo1 is upgraded.
4. The depEngine checks for cycled dependencies. This situation occurs when a package foo depends on package foo1, and vice versa.
5. The depEngine finds the dependency 'foo1 RO foo2' and sees 'foo2' is missing files.
6. The depEngine returns an error to the packagemanager.

This scenario follows the exception path 2. The depEngine checks for cycled dependencies and lists these packages and their dependency (BCoE: 5). The depEngine reports an error to the package manager.

1. The AbT-Linux package manager requests the dependency-tree uptree of a package 'foo1'.
2. The depEngine loads the dependency tree.
3. The depEngine looks at the dependency info of a package in the packages in the higher levels in the tree for the packages relying on 'foo1'. The depEngine will find the dependency 'foo RO foo1' and sets the rebuild-property for package foo to true when foo1 has an altered configuration for example. The depEngine searches the tree further down. The depEngine for instance processes 'foo DO foo1' and sets the rebuild-property for package foo to true because foo1 is upgraded.
4. The depEngine checks for cycled dependencies. This situation occurs when a package foo depends on package foo1, and vice versa.
5. The depEngine finds the dependency 'foo1 RO foo2' and finds the dependendcy 'foo2 RO foo1'.
6. The depEngine returns this cycled dependency to the packagemanager.

10.6 Request number of pkg's downtree dependencies

This scenario follows the basic course of events. The abt package manager needs to know the number of downtree dependencies of the package 'foo' to know the influence editing the package could have.

1. The abt package manager submits the package 'foo' to the depEngine for requesting the number of pkg's downtree dependencies.
2. The depEngine reads the package description for 'foo' to pick up the dependencies down tree.
3. The depEngine finds a dependent package 'foo1' in the description of 'foo'. The number of dependencies is now increased to 1.
4. The depEngine finds a dependent package 'foo2' in the description of 'foo'. The number of dependencies is now increased to 2.
5. The depEngine checks the description of the dependent package 'foo1'. It does not find any dependent package.
6. The depEngine checks the description of the dependent package 'foo2'. It finds the package 'foo3' in the description of 'foo2'. The number of dependencies is now increased to 3.
7. The depEngine checks the description of the dependent package 'foo3'. It does not find any dependent package.
8. This number (3) is returned to the package manager. The package manager knows that editing 'foo' will have influence on 3 dependencies downtree.

This scenario follows the alternative path. There is no package given to the depEngine.

1. The abt package manager requests the number of pkg's downtree dependencies. But, it forgets to submit the package of which the downtree dependencies must be counted.
2. The depEngine asks the abt package manager for a package.
3. The abt package manager submits the package 'foo' to the depEngine for requesting the number of pkg's downtree dependencies.
4. This use case is continued with the Basic Course of Events.

This scenario follows the exception path. Counting the dependencies somehow fail. The depEgine reports an error to the abt package manager.

1. The abt package manager submits the package 'foo' to the depEngine for requesting the number of pkg's downtree dependencies.
2. The depEngine reads the package description for 'foo' to pick up the dependencies down tree.
3. The depEngine finds a dependent package 'foo1' in the description of 'foo'. The number of dependencies is now increased to 1.
4. The depEngine fails in locating the package 'foo1'. It might have happened that the package got corrupted or erased somehow.
5. The depEngine returns the result by giving an error to the abt package manager.
6. This use case will be stopped by the depEngine.

10.7 Request buildorder

This scenario follows the basic course of events. The package manager wants a buildorder for building various version of packages that depend on each other. Hereby the dependencies must be followed without breaking them.

1. The abt package manager requests a proper build order for a package list (foo, foo1, foo2)
2. The depEngine takes the first package 'foo' and checks the package description of 'foo' for dependency info.
3. If in the dependency info of 'foo' a required version is found, 'foo v1.0 DO = foo1 v1.0', the depEngine checks if 'foo1' whether it is the right version.
4. If 'foo1' is the right version, the depEngine lists 'foo' and when no other dependencies for 'foo' are found, the depEngine continues looking at the dependency info of 'foo1'.
5. If the dependency is 'foo1 RO ¿ foo2 1.0', 'foo1' must always be rebuilt when the configuration of 'foo2' is changed and 'foo2' can be any version from 1.0.

6. If 'foo2' is not the right version, the depEngine reports this to the package manager.
7. If 'foo2' is one of the right versions, the depEngine places foo in the buildqueue.
8. If the dependency is 'foo1 DO foo1 ¡ 1.0', 'foo1' must be rebuild before 'foo' and 'foo1' can be any version of upto version 1.0.
9. The buildqueue could now be (foo,foo1,foo2)
10. Having ensured the correct versions are present, the depEngine checks what kind of dependency the packages have between eachother.
11. The depEngine takes the package 'foo' and checks the package description of 'foo' for dependency info.
12. When 'foo DO foo2', foo2 must go before foo.
13. The depEngine moves foo2 so the buildqueue is now (foo2, foo, foo1).
14. When no other dependencies for 'foo' are found the depEngine searches for the dependency of 'foo2'.
15. When 'foo2 DO foo1', 'foo1' must go before 'foo2'.
16. The depEngine moves foo1 so the buildqueue is (foo1, foo2, foo)
17. The depEngine returns this order to the package manager.

This scenario follows the basic course of events. The package manager wants a buildorder for building packages that rely on each other (Ro). Hereby the dependencies must be followed without breaking them.

1. The abt package manager requests a proper build order for package list (foo, foo1, foo2).
2. The depEngine takes first pkg 'foo' and checks 'foo' package description for dependency info.
3. When 'foo RO foo2', 'foo' must be always rebuilt when the configuration of 'foo2' is changed.
4. 'foo2' is listed first (foo2, foo, foo1)
5. When 'foo2 RO foo1', 'foo2' must always be rebuilt when the configuration of 'foo1' is changed.
6. The depengine moves 'foo1' so the buildqueue is now (foo1, foo, foo2)
7. The depEngine returns this order to the package manager.

This scenario follows the basic course of events. The package manager wants a buildorder for building packages that optionally rely on each other (oRo). Hereby the dependencies must be followed without breaking them.

1. The abt package manager requests a proper build order for package list (foo, foo1, foo2)
2. The depEngine takes first pkg 'foo' and checks 'foo' package description for dependency info.
3. When 'foo oRO foo2', when this dependency is chosen, there is extra functionality added and an RO dependency is added. 'foo' has to be changed when the configuration of 'foo2' is changed.
4. Then 'foo2' is listed first (foo2, foo, foo1)
5. When 'foo2 oRO foo1', there is extra functionality added and an RO dependency is added. 'foo2' has to be changed when the configuration of 'foo1' is changed. The package manager decides whether to chose the option or not.
6. The depengine moves 'foo1' so the buildqueue is now (foo1, foo, foo2)
7. The depEngine returns this order to the package manager.

This scenario follows the basic course of events. The package manager wants a buildorder for building packages that optionally depend on each other (oDo). Hereby the dependencies must be followed without breaking them.

1. The abt package manager requests a proper build order for package list (foo, foo1, foo2)
2. The depEngine takes first pkg 'foo' and checks 'foo' package description for dependency info.
3. When 'foo oDO foo2', there is extra functionality added and an DO dependency is added. 'foo2' must go before 'foo'.
4. The depEngine moves 'foo2' so the buildqueue is now (foo2, foo, foo1).
5. The depEngine searches for the dependency of 'foo2'.
6. When 'foo2 oDO foo1', there is extra functionality added and an DO dependency is added. 'foo1 must go before foo2'.
7. The depEngine moves 'foo1' so the buildqueue is (foo1, foo2, foo)
8. The depEngine returns this order to the package manager.

This scenario follows exceptionpath 1

1. The abt package manager requests a proper build order for package list (foo, foo1, foo2)
2. The depEngine takes first pkg "foo" and checks "foo's" package description for dependency info and tries to find the dependencies of "foo" but the dependency-entry for "foo" does not seem to be available.
3. The depEngine returns an error message to the package manager telling it the dependecy-tree is invalid.

10.8 Select level of logging

This scenario will follow the basic course of events. The developer of the package manager thinks that there might be an error somewhere, but does not know where. For better understanding of the situation the package manager is set to a higher logging-level.

1. The package manager selects the level 9 for the logging of the depEngine.
2. When the depEngine handles a request from the package manager, all actions will be logged according to the logging level selected.
3. The depEngine returns this log-file to the package manager.

This scenario will follow alternative path 1.

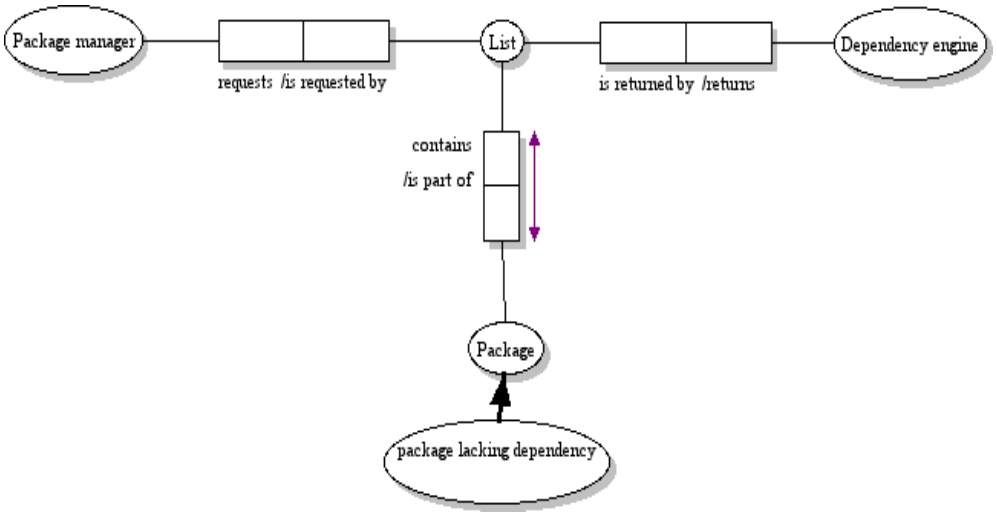
1. The package manager selects that actions don't have to be logged.
2. When the depEngine handles a request from the package manager, no action will be logged by the depEngine.

This scenario will follow alternative path 2.

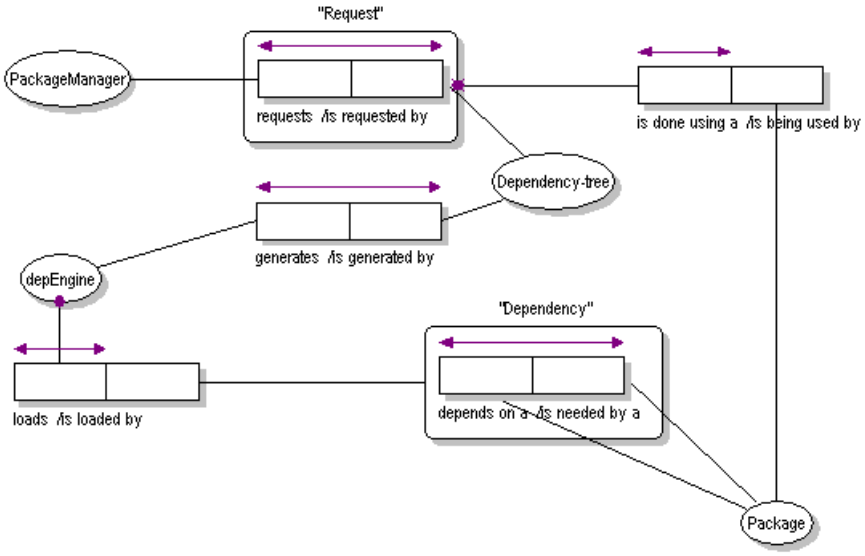
1. The package manager selects the level 9 for the logging of the depEngine.
2. When the depEngine handles a request from the package manager and tries to write the actions to the log-file it does not get write-permissions to the file.
3. The depEngine returns a message to the package manager it can't log the actions.

11 Domain Models

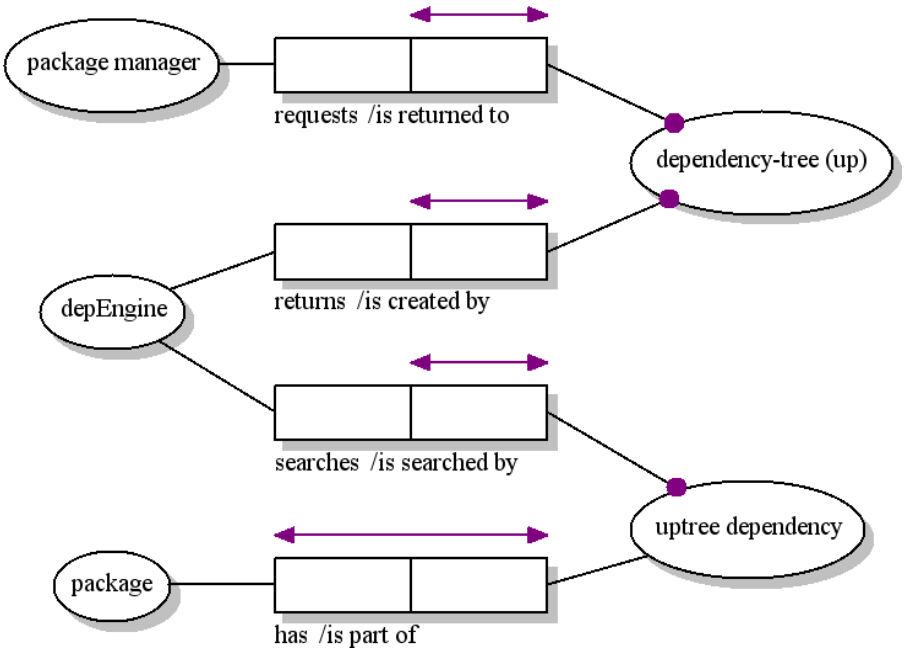
11.1 Request list of packages lacking dependencies



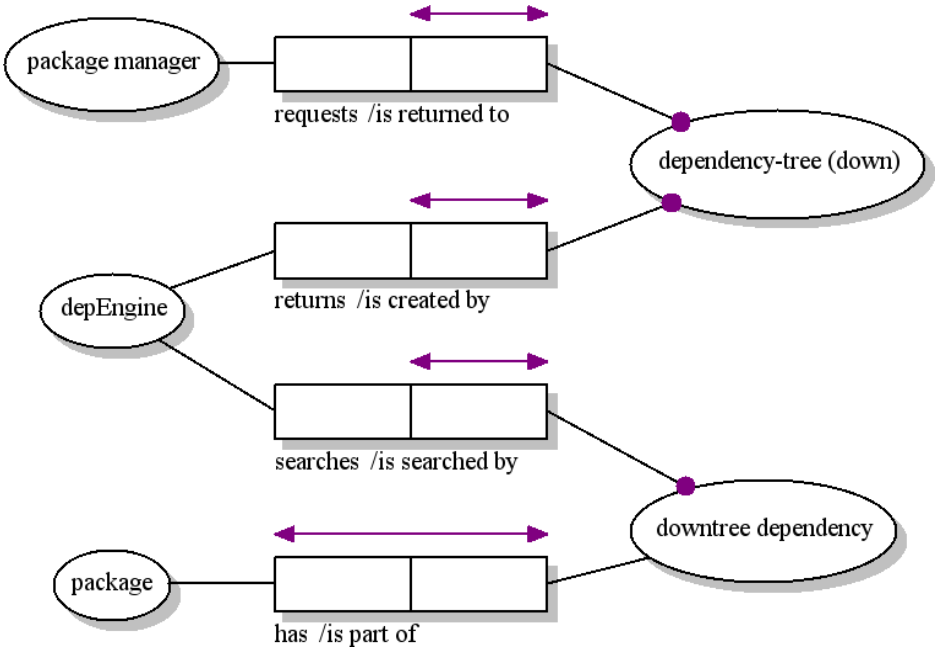
11.2 Request dependency-tree



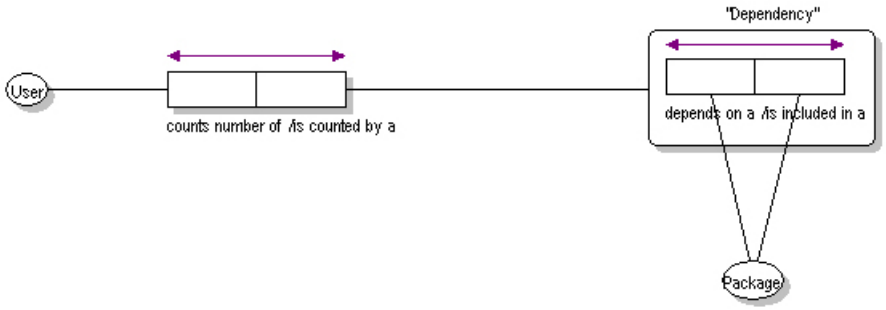
11.3 Request dependency-tree uptree



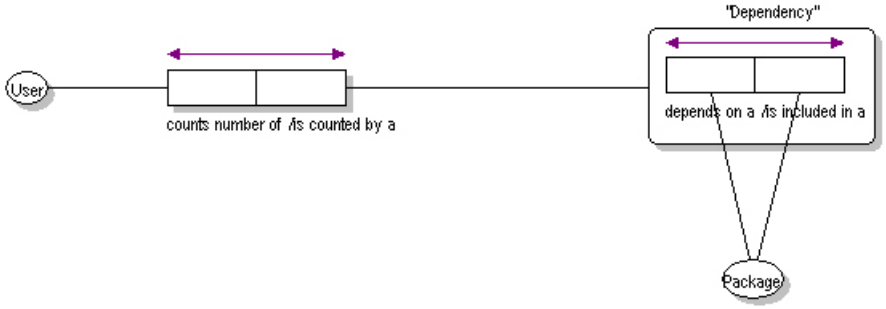
11.4 Request dependency-tree downtree



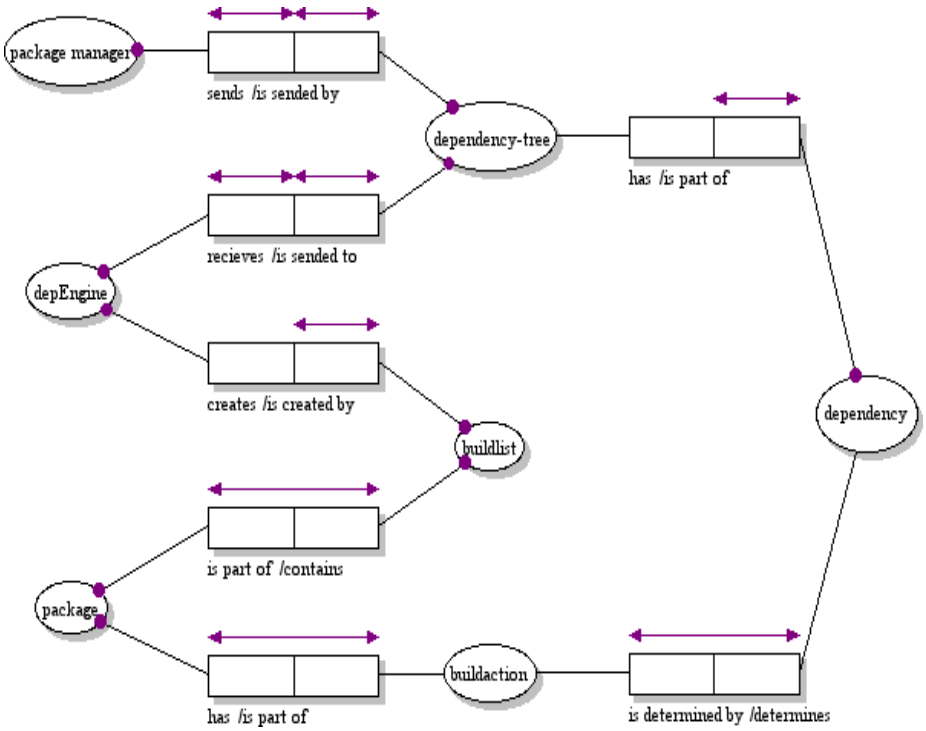
11.5 Request number of pkg's uptree dependencies



11.6 Request number of pkgs downtree dependencies



11.7 Request buildorder



11.8 Select level of logging

12 Business Rules

1. The depEngine and the the abt-package manager will be published under GPL license, because the executive sponsor wants to make a contribution to the Open Source community.

13 Non-functional Requirements

1. **Adaptivity:** Ability for the system to fit in as part of a larger system. The depEngine should be able to work with the abt package manager, which is a part of the AbT-Linux distro.
2. **Traceability:** All actions of the depEngine will be logged.
3. **Transparency:** As in all Open Source projects, transparency is important. The source code is available and users are free to make comments or suggests enhancements when they come up with any.
4. **Reliability:** The depEngine should be working as stated, without errors and accurate. The abt package manager should be able to trust on the build list generated by the depEngine.
5. **Performance:** The depEngine must perform it's tasks within an acceptable time-span. The executive sponsor will eventually define which time-span is acceptable.
6. **Security:** The source code that is in development should be secured so not everybody can moderate it in the CVS.

14 Terminological Definitions

1. **AbT-Linux:** AbT-Linux stands for About Time Linux. This is Lunix distribution Eric Schabell started.

2. **Abt package manager:** The abt package manager is a software package managing tool for managing packages on AbT-Linux. This is an internal software component to solve dependency questions.

3. **Build list:** A build list declares the order in which the linked packages should be build. It can be sorted or unsorted.

4. **Build queue:** The place where the build list is processed, as the build list could be unsorted and we hope after you sort it that the Build queue will then follow a correct order.

5. **Configure:** Configuring the pre-build phase of building a package. previous build and package is rebuilt.

6. **Dependencies:** Dependencies are links to packages that rely or depend on the current package.

7. **Dependencytree:** A dependencytree describes the dependencies between the different packages.

8. **Depends on (DO):** Depends on means that a package will be rebuild anytime the package it depends on is rebuild.

9. **Distribution:** A distribution is a collection of software under one name.

10. **Install:** Copy binaries to specified locations.

11. **LaTeX:** LaTeXis a high-quality typesetting system, with features designed for the production of technical and scientific documentation. LaTeXis the de facto standard for the communication and publication of scientific documents.

12. **Optionally Depends on (oDO):** Optionally Depends on means that a package will only be rebuild, anytime the package it depends on is rebuild, if the user permits it.

13. **Optionally Relies on (RO):** Optionally Relies on means that a package will only be rebuild if the configuration of the package it relies on changes and the user permits it.

14. **Package:** A package is a complete set of data, which the abt package manager can use to manage all aspects of using this single piece of software.

15. **Rebuild:** Rebuild is to recompile the software package, no options changed from the previous time a build was done.

16. **Reconfigure:** Reconfigure means that options are changed from the the previous time a build was done.

17. **Relies on (RO):** Relies on means that a package will be rebuild if the configuration of the package it relies on changes.

18. **Script/build script:** Automated build process for certain packages.

19. **Software package manager:** The software package manager is a software application that manages all aspects of handling dependencies of software during installation, upgrading and uninstalling packages.

20. **Source:** The source is the program code of package.