

# Package

## An interface to a software package.

### Table of contents

1 Overview.....	2
1.1 Intent.....	2
1.2 Problem.....	2
1.3 Discussion.....	2
1.4 Structure.....	3
1.5 Example.....	4
1.6 Check List.....	5
1.7 Rules of Thumb.....	5
2 Design.....	5
3 Commands.....	5
3.1 create.....	6
3.2 extract.....	6
3.3 finish.....	6
3.4 get.....	7
3.5 install.....	7
3.6 prepare.....	7
3.7 purge.....	8
3.8 register.....	8
3.9 remove.....	9
3.10 upload.....	9
3.11 verify.....	9

## 1. Overview

**Package:** *An interface to a software package.*

### 1.1. Intent

- Define an object that encapsulates the lifecycle of a software package.
- Abstract the parameters common to packages of different formats.

### 1.2. Problem

We want to create and install packages of various formats but these packages have varying degrees of capability and require their own tool sets to produce them. Lacking a standard way of creating, distributing and installing packages undermines the goal of a generic deployment procedure.

### 1.3. Discussion

Depending on their architecture and platform, applications are released using a wide variety of formats. The components of a Java based applications may be deployed as .jar, .war, and .ear files, while platform software may be deployed as .rpm, .deb, and .pkg files. Windows based applications have their own range of deployable artifacts and installers: .dll, .exe, .setup, .zip. Indeed, many software components are distributed as they are, devoid of any formalized packaged delivery.

When one considers the common details and requirements for software packages they can be organized into different categories:

- parameters: these are key attributes that describe the package.
- lifecycle: these are the set of procedures needed to create and manage the package.
- dependencies: these are the requirements needed both to build the package and the assumptions about the dependencies that will be met when the package is installed.

Due to this diversity of software delivery, one is often faced with using a myriad of tools and a recipe of procedures to prepare, deliver and install software components. This requires that the person undertaking the deployment must be well versed in the use of these tools and may also require that a developer maintain specialized locally developed code to produce the artifacts.

Having to install packages of differing formats to support a multi-component application undermines the desirable goal to employ a single common mechanism. A preferred scenario

of course, would be to have a homogeneous package format so that there could be a standard method to create and manage all packages.

Another approach would be to define a package concept that sufficiently describes the characteristics, both essential attributes and dependencies, as well as, encapsulate the lifecycle methods for the package.

This package abstraction offers the benefit of homogeneity by way of its standardized behavior and characteristics yet allows for the underlying differences of the heterogeneous formats and tools needed to support different application components. As new package formats are introduced, their differences can be accommodated by defining a new derived class.

The package abstraction organizes the details of the package life cycle into several categories: configuration, procedures, context and content.

Package configuration encompasses essential installation, version, dependency, storage and deployment attributes. This configuration data is used to drive the package life cycle procedures.

Life cycle procedures fall into two general categories: creation and installation. Creation methods take as input package content and output a package artifact using the format specific to that package type. Additionally, the creation method can store the package artifact into a release repository for later distribution. Installation methods pull the package from the repository, extract it to the host and perform optional pre- and post-installation.

Occasionally, environment specific information is required by the life cycle methods (though one should strive to avoid this), therefore the package abstraction provides the means to override configuration parameters with needed values.

While the package abstraction can be implemented to contain the content of the package archive itself, it is preferable to decouple the content file itself from the code that creates and installs it. With this arrangement the package archive file (e.g, the .rpm file) resides in a repository, while an instance of a package type, provides the configuration and environment context and procedures to install the package.

## 1.4. Structure

There are two primary collaborations of the package abstraction one for building the package and the other for deploying the package.

The Builder uses the Package type to construct, register and store the package. The Deployment object uses the Package to deliver and install the package.

The essential package properties are defined as the following parameters:

Name	Description
arch	Host architecture type.
base	The package base name. This often is the name of the package minus the file extension.
buildtime	Timestamp specifying when the content of the package was built.
install-rank	A value representing installation order.
release	The version release identifier.
restart	Boolean flag specifying that a service restart is required after installation.
release-tag	A logical release identification tag.
filetype	Package format type
install-root	Directory path where package should be extracted.
vendor	Organization responsible for creating the package
repo-url	URL used to access the package artifact from the repository.
version	Package version
filename	The name of the package file.

**Table 1: Parameters**

## 1.5. Example

The Package type defines an object that provides methods to create and install a package and maintain key properties that describe the package. Packaging and deploying the Apache httpd server is a good example to demonstrate the use of the Package type.

The graphic below describes the two phases of the build and installation cycle for a package. In the build phase, a Builder object takes source files and a build configuration and compiles and uses the Package's `create`, `upload` and `register` commands to store it in the repo. In the installation phase, a Deployment object uses the Package's `install` command which implicitly uses the `prepare`, `get`, `extract` and `finish` commands to acquire and

install the package, providing optional environment context during the process.

## 1.6. Check List

### Building

1. Choose the package format type required. This may be from existing package types like: zip, jar, rpm, etc.). If a new format is desired then derive a new subtype from Package and implement the necessary installation lifecycle methods.
2. Identify the set of files that will be archived using the above package format. Then run the [create](#) command specifying necessary parameters.
3. Store the package artifact in the repository and register it as a resource that can be used as Deployment dependency.

### Installation

1. Identify Deployment types that require this package type. Possibly, modify the dependency constraints for the Deployment type to allow the new dependency type.
2. Choose the Deployment object and Package object and assign the dependency.
3. Install the package by running the [Deployment](#) object's Packages-Install command.

## 1.7. Rules of Thumb

Since package formats and the tools that produce and manage them are not equal, one must establish a middle ground.

## 2. Design

### Super Type Managed-Entity

Role	<b>Concrete.</b> (Objects can be created.)
Instance Names	<b>Unique</b>
Notification	<b>false</b>
Template Directory	
Data View	Children, proximity: <b>1</b>
Logger Name	Package

## 3. Commands

**Note:**

Commandline options displayed in square brackets "[]" are optional. If an option expects arguments, then angle brackets are shown after the option "<>". Any default value is shown within the brackets.

**3.1. create**

*creates a package archive*

**Usage**

```
create [-filename <>] [-installroot <>]
```

**3.1.1. Options**

Option	Description
filename	<i>file to write archive</i>
installroot	<i>root where source files reside</i>

**3.2. extract**

*extracts the package*

**Usage**

```
extract [-base <>] [-filename <>] [-installroot <>]
```

**3.2.1. Options**

Option	Description
base	<i>package base name</i>
filename	<i>file to extract</i>
installroot	<i>destination directory</i>

**3.3. finish**

*finishes the package installation*

**Usage**

```
finish [-filename <>] [-filtersfile <>] [-installroot <>]
```

**3.3.1. Options**

Option	Description
--------	-------------

filename	<i>path to package file</i>
filtersfile	<i>property file</i>
installroot	<i>installation root dir</i>

### 3.4. get

*gets the package from the repository*

#### Usage

```
get [-filename <>] [-installroot <>] [-url <>]
```

#### 3.4.1. Options

Option	Description
filename	<i>the filename</i>
installroot	<i>the installation root</i>
url	<i>the url to the package</i>

### 3.5. install

*runs the package installation process*

#### Usage

```
install [-filename <>] [-filtersfile <>] [-installroot <>]  
[-url <>]
```

#### 3.5.1. Options

Option	Description
filename	<i>the filename</i>
filtersfile	<i>filtersfile</i>
installroot	<i>the install root</i>
url	<i>the repo url</i>

### 3.6. prepare

*prepares the package for installation*

#### Usage

prepare

### 3.7. purge

*removes the file from the repository*

#### Usage

```
purge -url <>
```

#### 3.7.1. Options

Option	Description
url	<i>URL of the file</i>

### 3.8. register

*registers the object*

#### Usage

```
register [-arch <>] [-base <>] [-buildtime <>] [-depot <>]
[-description <>] [-filename <>] [-filetype <>]
[-installrank <>] [-installroot <>] [-name <>] [-release <>]
[-releasetag <>] [-restart] [-type <>] [-url <>] [-vendor
<>] [-version <>]
```

#### 3.8.1. Options

Option	Description
arch	<i>host architecture</i>
base	<i>the package base name</i>
buildtime	<i>time package was built</i>
depot	<i>project depot name</i>
description	<i>description of the object</i>
filename	<i>the file name</i>
filetype	<i>the package file type</i>
installrank	<i>install rank</i>
installroot	<i>the installation root</i>
name	<i>the object name</i>

release	<i>the package release</i>
releasetag	<i>release tag</i>
restart	<i>requires restart</i>
type	<i>the object type</i>
url	<i>the repo url</i>
vendor	<i>the package producer</i>
version	<i>the package version</i>

### 3.9. remove

*removes the installed package*

#### Usage

`remove [-installroot <>]`

#### 3.9.1. Options

Option	Description
installroot	<i>directory to remove</i>

### 3.10. upload

*uploads the file to the repository*

#### Usage

`upload [-filename <>] [-url <>]`

#### 3.10.1. Options

Option	Description
filename	<i>file to upload</i>
url	<i>repository url</i>

### 3.11. verify

*verifies the package*

#### Usage

`verify`

