PHP PEAR Packages

Lecture-6

By: Ashiqullah Alizai
Herat University
Computer Science Faculty

Alizai.csf@hotmail.com

Introduction

- Efficient handling of code can make a big difference;
- Example:
  - when administrating more than one PHP application
  - several installations of the same application,
  - to aid the distribution of several PHP applications using the same base libraries.
- Efficient code reuse is not a new idea
- Various degrees of efficient code reuse can be achieved through:
  - Libraries,
  - Classes,
  - object oriented programming,
  - Etc.
- An important aspect of code reuse is the management and accessibility of these libraries.

What Is a package?

- A package is a name-space that organizes a set of related classes and interfaces
- Conceptually you can think of packages as being similar to different folders on your computer.
- When it comes to organizing, reuse and share your code, packages are a great way to allow you to do this.
- They can contain all sorts of code like
  - Models,
  - Third-party libraries,
  - Configuration Files
- Packages also allow you to extend the core without messing up your app/classes directory.

Concept of Packages

- A logical group of libraries with common functionality.
- This does not necessarily mean that code packages have to be dependent one upon the other, although packages can be defined which require other packages.
  - For example, it may not make much sense clumping together as one package a set of text formatting libraries with a set of file system libraries.
- However, if necessary, one could be defined as dependent upon the other and kept as two separate packages.

Software Package

- A software package is a group of programs which are bundled together to serve a common purpose.
- Software package include
- the source code that built the executable programs a variety of documentation for the programs themselves.
- Some software packages will also include example files that can further illustrate how the other components of the package work.
- The components of a software package can do significantly different things, but all the components of the package come together in a unified whole.
- Some software packages have one main program that encapsulates all the smaller programs.

Pear Packaging

- PEAR is a good example of reusable PHP code; its name is an acronym for "PHP Extension and Application Repository".
- PEAR is a collection of packages, and is installed in a central location accessible to any PHP code running on that machine.
- The emergence of Pear
  - During version 4 of PHP, the number of users exploded, and so did the number of code snippets you could download from different web sites.
  - Some of these sites offered code that you had to copy and paste into your editor, while others let you download archives with source files.
  - This was useful to many people, but there was a need for a better way of sharing and re-using PHP code, similar to Perl's CPAN.
- The PEAR project set out to solve this problem by providing an installation and maintenance tool and code/release management standards.

Pear Packaging

- The purpose of PEAR to set:
  - A structured library of open-source code for PHP users
  - A system for code distribution and package maintenance
  - A standard style for code written in PHP, specified here

Advantages of PEAR Packages

- Advantage of library packaging
  - The simplification of product release management
  - Separating out the core libraries from a product, and releasing them as packages, means that a single bug fix in a library will not prompt the generation of a whole new product release;
  - Packaging requires some thoughtful division of functionality, so the end result could actually give you the hidden bonus of more portable and cleaner code.

Pear services

- PEAR provides
  - The PEAR Installer (a package-management tool)
  - Packages with PHP library code
  - Packages with PHP extensions (PECL)
  - PEAR coding standards, including a versioning standard

A spin-off from the PEAR project

- **PECL**, the PHP Extension Community Library.
- PECL used to be a subset of PEAR, but today, it is managedseparately. This means that PECL has its own web site, mailing lists, administrativeroutines, and so on.
- However, PEAR and PECL share tools and infrastructure: Both use the PEAR Installer, both use the same package format, and both use the same versioning standard.
- The coding standard is different however: PECL follows the PHP coding standard (for C code), while PEAR has its own.

PEAR CONCEPTS: packages

- When you want to install something from PEAR, you download and install a particular release of a **package**. (You learn more about releases later on.)
- Each package has some information associated with it:
  - Package name (for example,HTML_QuickForm)
  - Summary, description, and home page URL
  - One or more maintainers
  - License information
  - Any number of releases
- PEAR packages are unlike other package formats, such as Linux's RPM, Debian packages, or the System V UNIX PKG format.
- One of the major differences with most of these is that PEAR packages are designed to be platform-independent, and not just within one family of operating systems, such as System V or Linux.

PEAR CONCEPTS: Releases

- As with PHP itself, the code that you actually install is packaged in a tar.gz orzip file along with installation instructions.
- PEAR packages can also be installed and read the installation instruction by PEAR Installer.
- In addition to this package-specific information, each release contains
- A version number
- A list of files and installation instructions for each
- A release state (stable, beta, alpha, devel, or snapshot)

PEAR CONCEPTS: Releases

- When you install a PEAR package, you receive the latest stable release by default, for example:
  - **$pear install XML_Parser**
  - downloading XML_Parser-1.1.0.tgz ...
  - Starting to download XML_Parser-1.1.0.tgz (7,273 bytes)
  - ....done: 7,273 bytes
  - install ok:
  - XML_Parser 1.1.0
- By running the command pear install XML_Parser , you obtain the latest stable release of the XML_Parser package, with the version number 1.1.
- There are several reasons why PEAR did not use an existing format such as RPM as its package format.
- The most obvious reason is that PHP is very portable, so the package format would have to be supported on every platform PHP runs on.

PEAR CONCEPTS: Version Numbers

- PEAR defines some standards for packages, "Building PEAR Components," and a versioning standard.
- The **versioning standard** tells you how to interpret a version number and, more importantly, how to compare two version numbers.
- PEAR's version number standard is pretty much what you are used to from open-source packages, but it has been put in writing and implemented through PHP's version
- __compare() function.
- **10.2.3.1 Version Number Format**
- A version number can be everything from a simple "1" to something awful, like "8.1.1.2.9b2." However, PEAR cares about at most three numbers, plus an extra part at the end reserved for special cases, like "b1," "RC2," and so on. The syntax is like

```
Major [ . minor [ . patch ]] [ dev | a | b | RC | pl [ N ]]
```

PEAR CONCEPTS: Version Numbers

- Most PEAR packages use the two- or three-number variation, sometimes adding a "release state" part, such as "b1," during release cycles.
- Here's an overview of the meaning of the release state component.

Table 10.1 Example Version Numbers

Version String	Major Version	Minor Version	Patch Level	Release State‘
1	1	—	—	—
1b1	1	—	—	b1
1.0	1	0	—	—
1.0a1	1	0	—	a1
1.2.1	1	2	1	—
1.2.1dev	1	2	1	dev
2.0.0-dev	2	0	0	dev
1.2.1RC1	1	2	1	RC1

PEAR CONCEPTS: Version Numbers

- Release state Example

Extra	Meaning
Dev	In development; used for experimental releases.
A	Alpha release; anything may still change, may have many bugs, and the API not final.
B	Beta release; API is more or less stable, but may have some bugs.
RC	Release candidate; if testing reveals no problems, an RC is re-released as the final release.
Pl	Patch level; (not very often) used when doing an "oops" release with last-minute fixes.

PEAR CONCEPTS: Comparing Version Numbers

```
Major [ . minor [ . patch ]] [ dev | a | b | RC | pl [ N ]]
```

- PEAR sometimes compares two version numbers to determine which signifies a "newer" release.
- For example, when you run the pear list-upgrades command, the version numbers of your installed packages are compared to the newest version numbers in the package repository on pear.php.net.
- This comparison works by comparing the major version first. If the major version of A is bigger than the major version of B, A is newer than B, and vice versa.
- If the major version is the same, the minor version is compared the same way. But as specified in the previous syntax, the minor version is optional so if only B has a minor version, B is considered newer than A.

PEAR CONCEPTS: Comparing Version Numbers

- If the minor versions of A and B are the same, the patch level is compared in the same way.
- If the patch level of A and B are equal, too, the release state part determines the result.
- The comparison of the "extra" part is a little bit more involved because if A is missing a release state, that does not automatically make B newer.
- Release states starting with "dev," "a," "b," and "RC" are considered older than "no extra part," while "pl" (patch level) is considered newer.

PEAR CONCEPTS: Comparing Version Numbers

- ***Major Versus Minor Version Versus Patch Level***
- So, what does it mean
- when the newest release of a package has a different major version than the one you have installed?
- Well, this is the theory: It should always be safe to upgrade to a newer patch level within the same major.minor version.

Version A	Version B	Newest?	Reason?
1.0	1.1	B	B has a greater minor version.
2.0	1.1	A	A has a greater major version.
2.0.1	2.0	A	A has a patch level; B does not.
2.0b1	2.0	B	A "beta" release state is "older" than no release state.
2.0RC1	2.0b1	A	"Release candidate" is newer than "beta" for the same major.minor version.
1.0	1.0.0	B	This one is subtle, adding a level makes a version newer.

PEAR Package Definition

- PEAR uses a file named *package.xml* for defining the structure of the package and associated files.
- It is an XML file and follows a specific format. The start to the file gives some instructions on how the file will be structured.
- These are pretty much fixed and should be the same for every package:

Distribution of Pear packages

- **A PEAR package is distributed as a gzipped, Tar, files**
- **Each archive consists of source code written in PHP, usually in an Object- oriented style.**
- **Many PEAR packages can readily be used by developers as ordinary third party code via simple include statements in PHP.**
- **PEAR package manager which comes with PHP by default may be used to install PEAR packages.**
- **PEAR packages do not have implicit dependencies so that a package's placement in the PEAR package tree does not relate to code dependencies.**
- **PEAR packages must explicitly declare all dependencies on other PEAR packages.**

PEAR package manager

- **The PEAR package manager provides an easy way to install, uninstall, or upgrade with new PEAR packages or PECL extensions.**
- **Before installing a package it can also be instructed to take care of package dependencies so all the extra needed packages are installed too.**
- **The PEAR package manager is run from the command line using the pear command.**
- **On PHP installations running on Linux, the PEAR package manager is ready for usage by default,**
- **Windows the PEAR package manager is only available after running a batch file called go-pear.bat.**

PEAR Package Definition Package.xml

- <?xml version="1.0" encoding="ISO-8859-1" ?>
  <!DOCTYPE package SYSTEM
  "http://pear.php.net/dtd/package-1.0">
  <package version="1.0">
- The DOCTYPE tag specifies the DTD structure of this XML file. in which you can see all the possible elements and attributes.
- We shall first take an overview of the structure of this XML file. Depending on how you plan to use your packages, not all of the following elements will be required, but it is a good idea to get acquainted with what is available:

Definition.....

- **package –** defines the package with a required version attribute.
  - **name –** the name of the package.
  - **summary –** a one line summary of the package.
  - **description –** a longer description outlining what the package is about.
  - **license –** the license under which this package is released.
  - **maintainers –** a list of persons associated with the package.

Definition.....

- **maintainer –** details about a person associated with the package.
  - **user –** a username to associate with this maintainer.
  - **role –** what this maintainer's role is with the package, eg. lead, developer, etc.
  - **name –** the maintainer's full name.
  - **email –** a contact email address for the maintainer.
- **release –** this is the main section defining the current release of the package.
  - **version –** the current version of this package.
  - **state –** allows for a state to be assigned and then filter for it during PEAR package operations, currently recognized states: stable, beta, alpha,

Definition.....

- **file –** a definition of a file which is part of this directory, we shall look at its attributes later.
  - **dir –** defines a new directory as part of the package and can contain further nested directory and file definitions.
- **deps –** a list of dependencies for this package.
  - **dep –** a dependency definition which we will examine in detail later.

The <filelist> Section

- This section is important to the whole package definition, as it describes where the package files are to be installed.
- The way that you subsequently include your class files in applications would obviously need to match the way this section is defined.
- The sub-element of the <filelist> section can have any combination of <file> and <dir> elements, and include nesting of one or more <dir> or <file> elements within a <dir> element. For example, the following describes a file, *Widget.php*, that is part of this package

```
<filelist>

  <file role="php" baseinstalldir="/">Widget.php</file>

</filelist>
```

The <filelist> Section

- php – a regular PHP source file which would be installed to the PEAR include directory.
- ext – a PHP extension file which would be installed to the PHP extensions directory or to the directory specified by PHP_PEAR_EXTENSION_DIR setting.
- doc – a documentation file, installed to [PEAR docs directory]/PackageName/
- test – a test file, installed to [PEAR tests directory]/PackageName/
- data – a data file, installed to [PEAR data directory]/PackageName/

The <filelist> Section

- script – a script used with this package which would be installed to the PHP binary directory or to the directory specified by PHP_PEAR_BIN_DIR setting.
- src/extsrc – C or C++ source which is not installed itself but is used to build an extension.
- The location of php role files can be further fine-tuned using the baseinstalldir attribute
- In the above example, the file *Widget.php* is to be installed in the root of the PEAR directory.
- You could specify any other directory in this attribute, relative to the root.
- Note that when nesting files within a <dir> section you can imply some <file> attributes by stating them in the <dir> element

```
<filelist>
  <file role="php" baseinstalldir="/">Widget.php</file>
```

Example

```
    <dir role="php" baseinstalldir="/Widget">

      <file>foo.php</file>

      <file>bar.php</file>
  </dir>

  </filelist>
```

The <deps> Section

- Although the idea may be to create self-enclosed packages of functionality, package code often relies on certain external functions or factors. You can specify not only other packages as dependencies, but a vast range of conditions.

Consider the following example of hypothetical dependencies for our Widget package:

```
<deps>

    <dep type="php" rel="ge">4.3.0</dep>
```

Example

```
    <dep type="ext" rel="has">gettext</dep>

<dep type="pkg" rel="has"
 version="1.2">HTTP</dep>

 </deps>
```

The <deps> Section

- What this section is stating is that the Widget package has to have a PHP version greater than or equal to 4.3.0, PHP has to have the get text extension, and a specific version, 1.3, of the HTTP package needs to be installed.
- There are several possible values for the type attribute; however, the three types primarily used are:
  - pkg – the dependency is another PEAR package.
  - php – the dependency is a specific PHP version.
  - ext – the dependency is a PHP extension.

The <deps> Section

- The rel attribute defines the relationship that is to be used for the check, and the following values are available:
  - has – contains.
  - eq – is equal to.
  - lt – is less than.
  - le – is less than or equal to.
  - gt – is greater than.
  - ge – is greater than or equal to.
- The <deps> tag could also have an additional attribute optional which, if left out, is assumed to have a value of “no”. If the value “yes” is specified, the dependency is no longer absolutely required, but only a recommendation which may enhance the functionality or features of the package.

The <deps> Section

- Check that you have included all your dependencies.
- As good practice, ensure also that they have been reduced to the necessary minimum.
- Look through your code and see if any dependencies could be resolved differently, perhaps by structuring packages in a different way, or by using a different approach.
- You should finally make sure that your *package.xml* file is valid.
- Fortunately PEAR has a utility within its pear command to do just that. Run the following code on your newly created *package.xml*:
  - pear package-validate package.xml
- If any errors are reported go back to your *package.xml* file and fix them, otherwise you are all set to install your package.

Installation

- Now that you have a valid package you can again turn to the pear command for installation:
  - pear install package.xml
- This will use the information in the *package.xml* file to install your libraries to the default PEAR directory. You can add some variations to this command using some extra flags:
  - **force –** to force an installation of a package ignoring any existing versions already installed. This is useful in situations when a package is updated but the version numberis not incremented.
  - **nodeps –** do not check for dependencies when installing this package and just install this single package.
- As a further note, you can automate the install of a large number of packages using a simple script to crawl a set of directories and execute pear install on each *package.xml* it finds. For an example of such a script, look at how this was handled by Horde at **http://cvs.horde.org/co.php/framework/install-packages.php**

Installation

- Since the PEAR directory is by default in the PHP include path, you can easily include your custom PEAR package in your applications as follows:

```
<?php
require 'Widget.php';
?>
```

This is the end for this lecture