A Package Management System for Web Based Applications

by

Edmund Chou

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science

and

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2003

Author .....
Department of Electrical Engineering and Computer Science
May 21, 2003

Certified by .....
Harold Abelson
Class O... MacVicar Teaching Fellow
...ervisor

Accepted by .....
. Smith
Chairman, Department Committee on Graduate Students

A Package Management System for Web Based Applications

by

Edmund Chou

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2003, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science
and
Master of Engineering in Computer Science and Engineering

Abstract

As web sites are increasingly used to provide complex functionality through web programming, this thesis offers a solution through which modular components known as packages provide specific features. The entire system is managed through the organization and configuration of individual packages in a modular fashion and promoting the reusability of code components. With the additional use of a package repository and publicly consumable web services, package management systems immediately receive current information about latest releases and user comments and feedback from a centralized location. Developers submit completed packages to the repository so that interested site administrators can be notified of the new addition ready for consumption. Those interested in a package also participate on a discussion forum with a web service interface, allowing both developers and administrators to easily share feedback and comments. The result is that developers can easily receive constructive feedback from several users to aid in the continuous improvement in released versions of the software. By forming a united community from what would normally be a disjoint group of developers and administrators, participants in the package management system have access to the information and knowledge of other peers allowing for a more productive environment.

Thesis Supervisor: Harold Abelson
Title: Class Of 1922 Professor and MacVicar Teaching Fellow

Acknowledgments

I would like to acknowledge the guidance I received in writing this thesis from Prof. Hal Abelson who offered his invaluable advice. Credit is also due to David Mitchell and Eric Carlson for their visions in project management which helped me with my feature design. Hal, Dave, and Eric all have had a significant role in every positive experience associated with the iCampus Project.

I would also like to thank Al Essa, Andrew Grumet and Tracy Adams for their constant support throughout the design and development of this project. They have given me the rare opportunity to work on the starting phases of a project with the potential to grow in to something very special, and for that I am grateful.

Finally on a more personal note, I have always believed that my friends and family have a profound effect on the sculpting of my personality over the years of my life. To my father, mother and sister who have always had faith in my abilities, to my friends who taught me the things I could not learn just by attending class, I thank you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The growth of the Internet has resulted in the birth of services and functionality offered by several web sites with an overlapping set of features. Despite multiple web sites requiring the use of similar systems such as message boards or calendars, new code is often written by web software programmers to replicate a feature previously developed elsewhere. Needless to say, the time and resources of talented programmers could be saved if web sites were able to collaboratively share a set of functionality by reusing completed code that has been thoroughly tested by a community of users.

This situation would greatly benefit from encapsulating individual features as packages and using a central package manager to fully configure and maintain a web site. Having a notion of packages providing common features to a web site would allow administrators to quickly and easily modify the site structure with minimal knowledge of the underlying code. With the added flexibility from the package management system, administrators can effortlessly customize aspects of the behavior and layout of their sites, and the modular design limits the effect of modifications to individual components of the site rather than affecting a monolithic structure in its entirety.

Furthermore, common packages could be distributed and incorporated in to other web sites with minimal effort as well. The model of wide distribution and usage of single components allows developers to create a single package serving functionality to be duplicated at multiple locations. Employing the principles of code reusability and modularity, package developers can collaboratively participate in creating a network

of features in an efficient and effective manner.

The goal of this thesis is the development of a package system capable of effectively reusing and managing software in web based applications. While a significant portion of this thesis is focused on the process of package development and the design of the local management system, an innovative use of packages at the repository level and web services makes this system unique with respect to existing package management systems. The introduction of web services to a package system integrates all the aspects of the system and forms a community focused on the development, usage and discussion of packages leading to code being widely used, thoroughly tested and reviewed, and further improvement. The entire system described will be used in conjunction with the iLearn system under the iCampus [9] research alliance between MIT and Microsoft Research.

Chapter 2 introduces the benefits of the package management system and the scenario for users of the system. Chapter 3 gives an overview of the problem by describing the design goals and implementation for the iLearn package management system. Chapters 4 and 5 provide greater detail about the features and designs of the package management system and package repository respectively. Chapter 6 compares the resulting system with the designs of related work and Chapter 7 describes future steps and possible improvements, particularly with the services supported by the package repository. Finally, Chapter 8 will conclude this thesis.

Chapter 2

Scenario

The encapsulation of web site functionality in to an individual package allows the application of code reusability to benefit web site administrators and developers. This chapter focuses on the specific usage scenarios of the package management system, demonstrating the power and importance of such a system for site administrators and developers, as well as how the repository based on web services can further enhance the end user experience.

2.1 Site Administration

Site administrators face the problem of being unable to effectively specify the desired behavior and functionality associated with their managed sites. They want a system that allows them the ability to configure the available features of a web site, with out modifying the associated code or causing massive disruptions to their site. By taking advantage of previously developed web site components in the form of packages, site administrators can solve this problem by dynamically adding, modifying or removing functionality through a simple web based user interface. Additional features of the package management system such as path resolution, contextualization and configuration settings allow site administrators the flexibility they need.

Using the web based user interface, a site administrator starts at a default page displaying a summarized list of installed packages. Consider an administrator who is

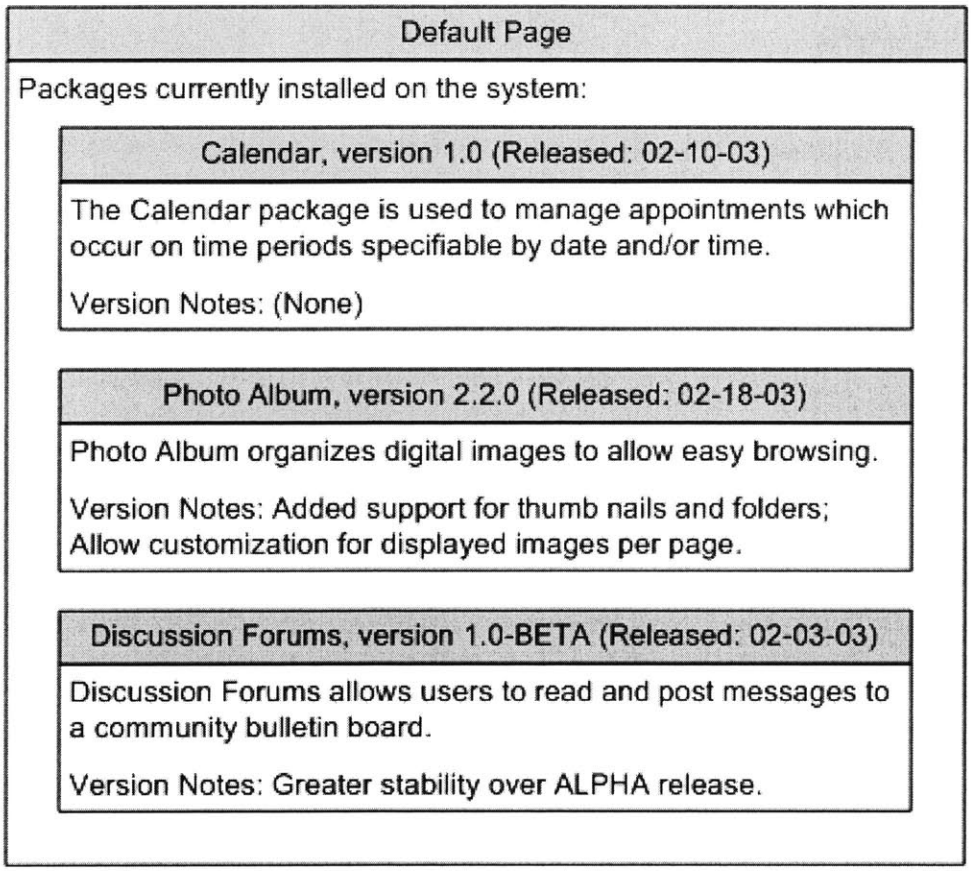

Figure 2-1: Package Manager Web Interface Default Page

managing a simple community site focused on sharing and discussing digital photography images. The administrator has a site with packages for calendars, discussion forums, and photo albums as illustrated in the sample default page in Figure 2-1. The purpose of the default page is to summarize the current state of the system, and the features available at the site.

2.1.1 Installation and Upgrade

To extend the available functionality available at the photography community site, the site administrator can either install a new package or upgrade an existing package. The simplest case for these procedures is where the administrator uploads a new package file to the system for installation or upgrade. However, this use case requires the administrator to have knowledge of the new packages and furthermore

had previously downloaded the relevant package files. This is a drawback to several package management systems, but the introduction of web services at the package repository will create a more usable environment.

The repository provides the additional benefit of allowing administrators to use web service queries in order to obtain lists of available packages for installation or upgrade. Figure 2-2 shows a sample full repository listing of package summaries, and Figure 2-3 shows a listing of summaries indicating which packages may be upgraded. From these pages, the site administrator can select a package then have the package manager retrieve the file from the repository and appropriately start an installation or upgrade process. Although the available functionality on the package upgrade page is a subset of that on the full repository package listing, the upgrade page provides a more concise view of packages likely to interest the site administrator.

Suppose the site administrator follows the appropriate links to install or upgrade additional functionality on the site. After installing a Community Directory package and upgrading the Discussion Forums package to the release version, the default page will display the new state as illustrated in Figure 2-4. A new entry for the installed Community Directory package is now included, and the new version of the Discussion Forums package is reflected in the list of packages.

2.1.2 Context Creation and Path Resolution

Although several packages are present on the site, contexts must be created and mounted for the packages to be publicly viewable. A context or instance of a package is an isolated usage of the package's functionality independent of the data and configuration settings associated with other contexts. By mounting a context at a named URL, visitors to the site will be able to view package contexts derived from intelligible names through the path resolution mechanism.

Table 2.1 shows a sample site map that would associate URLs with the specific package contexts. Mappings can be created automatically during various events or manually by site administrators. For example, in the site map given in Table 2.1, the user registration process follows the default behavior of creating a single context

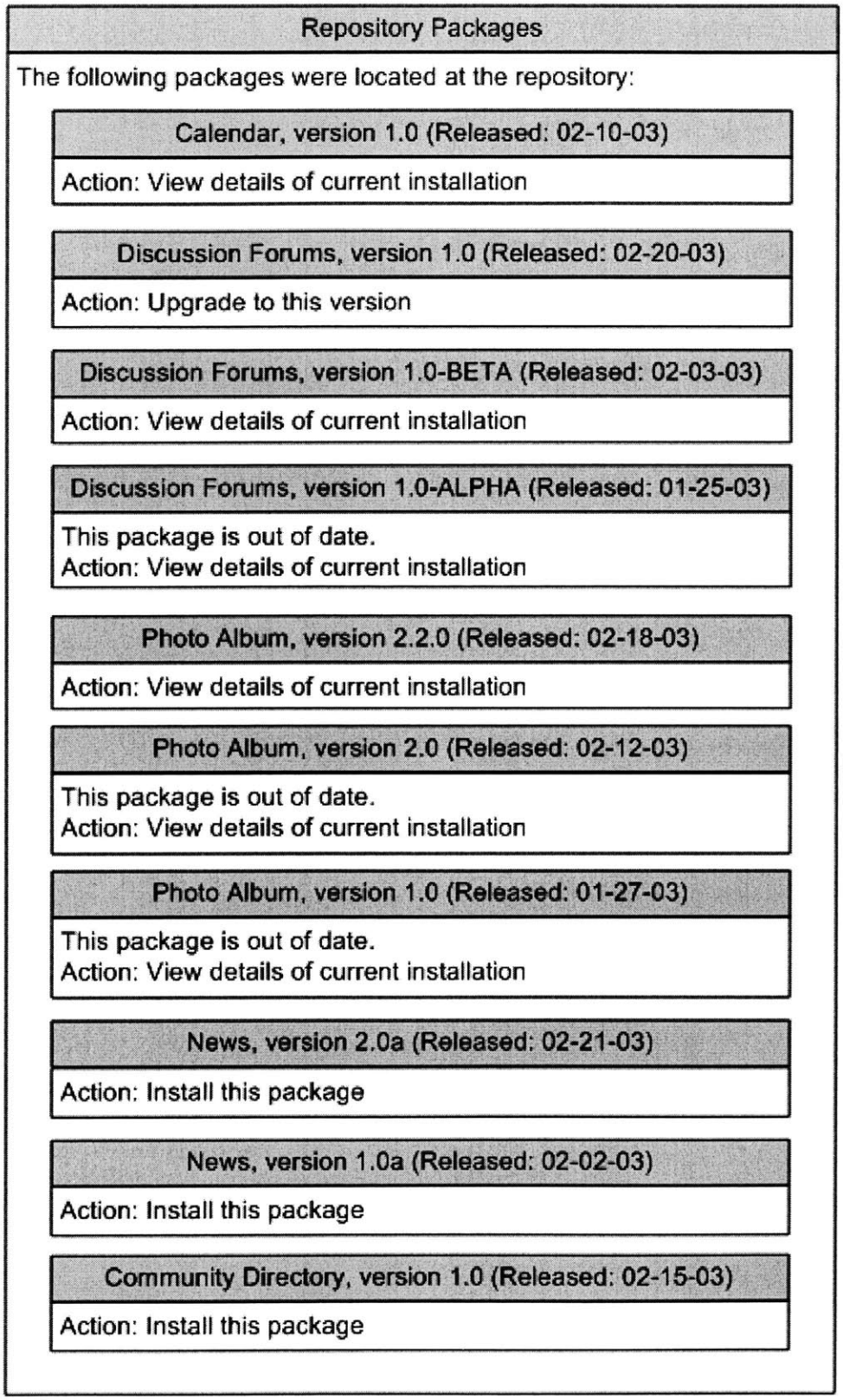

Figure 2-2: Package Manager View of Repository Packages through Web Services

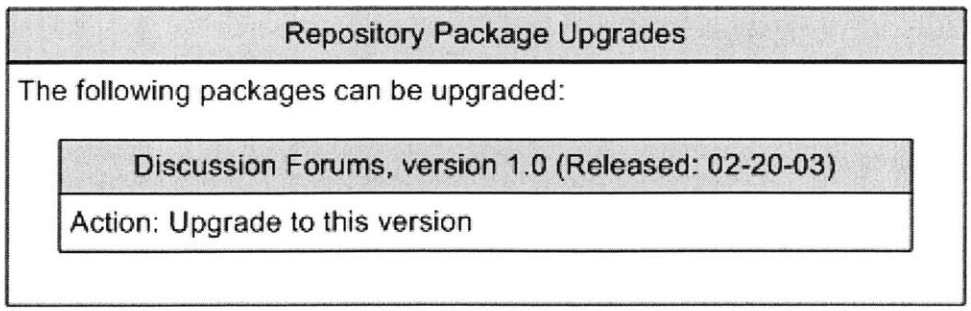

Figure 2-3: Package Manager View of Repository Package Upgrades through Web Services

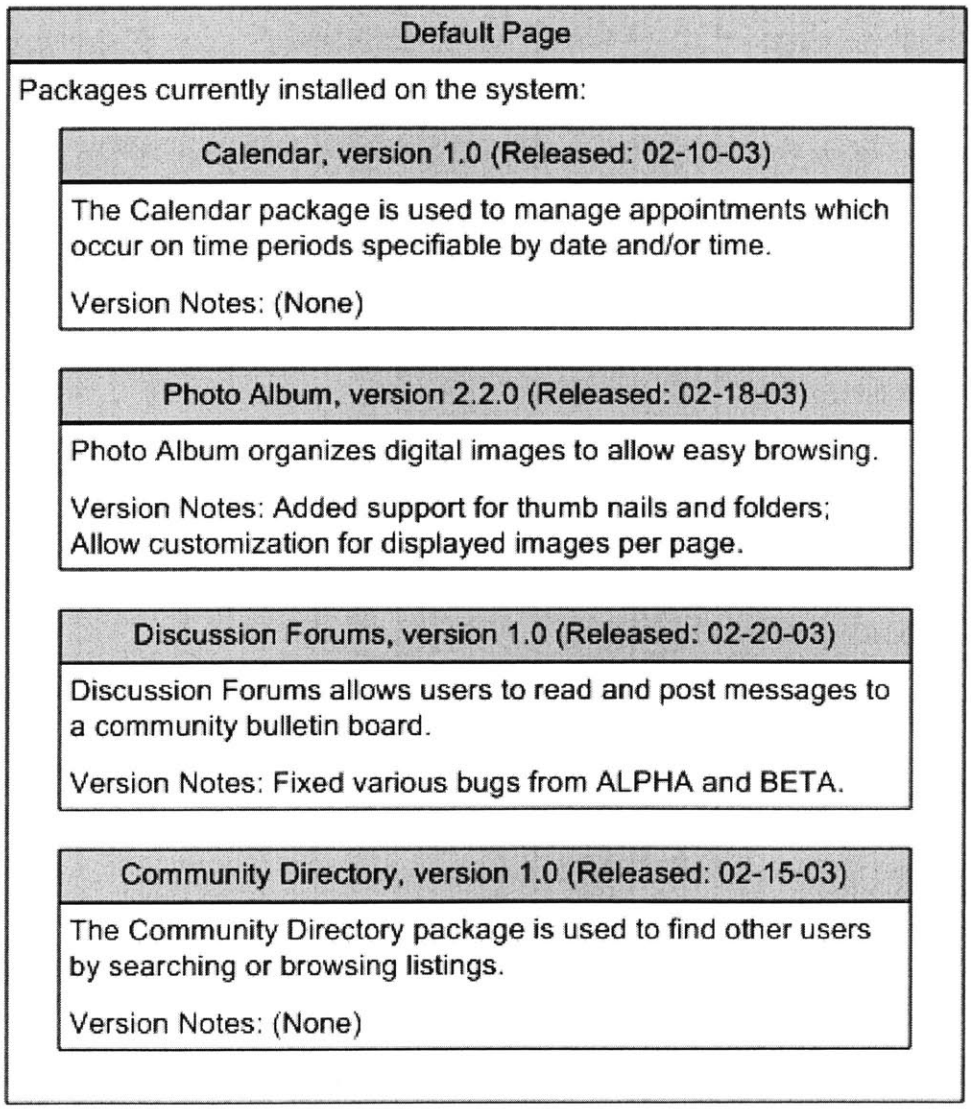

Figure 2-4: Package Manager Web Interface Default Page after Install and Upgrade

Context ID	Package Name	Context Name	URL
100	Discussion Forums	Photo Discussions	/photo-forum
101	Discussion Forums	Equipment Comparison	/equipment-forum
102	Discussion Forums	Miscellaneous	/misc-forum
103	Photo Album	Alice's Photo Album	/alice/photos
104	Photo Album	Bob's Pictures	/bob/photos
...	...	...	...
200	Calendar	Alice's Calendar	/alice/calendar
201	Calendar	Bob's Calendar	/bob/calendar
...	...	...	...
300	Community Directory	Directory	/users

Table 2.1: Site Map

for each of the Photo Album and Calendar packages and mounting them under a URL derived from the name of the user. However, a web user with the required authorization, presumably the site administrator, could edit the mapping to use a different URL, or create an additional mapping used as an alias to a context. Finally, a context can be unmapped and removed if, for example, a user decides that he does not require usage of the Calendar package.

2.1.3 Contexts and Settings

Package contexts can be used to customize the feature usage granularity on the site by specifying the number and location of mounted packages. An additional capability in the contextualization of packages is the customized settings unique to each context, as shown in the sample settings in Table 2.2. The site administrator has modification access to the settings associated with each context, allowing for customized behavior in the various parts of the site.

For example, of the three Discussion Forum package contexts, the context for photography discussions has been configured to be moderated since the administrator wants to protect against the possibility of users being overly harsh and critical of other works. Other customizations are simply based on user preferences, as shown in the separate contexts for user photo albums. While Alice prefers the use of thumbnails

Context ID	Context Name	Setting Name	Is Global?	Setting Value
100	Photo Discussions	Is Moderated?	False	True
101	Equipment Comparison	Is Moderated?	False	False
102	Miscellaneous	Is Moderated?	False	False
103	Alice's Photo Album	Show Thumbnails	False	True
103	Alice's Photo Album	Images Per Page	False	10
103	Alice's Photo Album	Local Directory	True	photo-uploads\
104	Bob's Pictures	Show Thumbnails	False	False
104	Bob's Pictures	Images Per Page	False	100
104	Bob's Pictures	Local Directory	True	photo-uploads\
...	...	...	...	...

Table 2.2: Configurable Package Context Settings

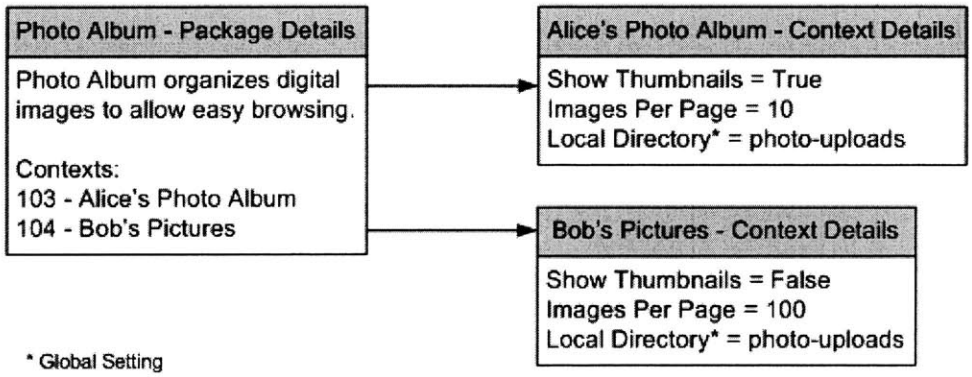

Figure 2-5: Package Manager Web Interface for Context Setting Configuration

for browsing images and displaying only a small number of images per page, Bob would rather have many images per page but without the use of thumbnails.

Also note that configuration settings can be global, usually for the case of system wide settings for a package. This is also illustrated in Table 2.2 where the Photo Album package requires a configuration setting for the local directory where uploaded images will be stored relative to. As shown in the example, uploaded images are stored relative to the local directory named `photo-uploads` and changes made to this configuration setting will affect all other contexts as well.

The web interface for this feature is illustrated in Figure 2-5 where the package details page has a list of contexts and links to configure the individual contexts.

2.1.4 Uninstall

Now consider the situation where the administrator realizes that members of the photography web community rarely use the functionality supplied in the Calendar package. Instead of individually removing each of the package contexts on the site, the site administrator can simply uninstall the Calendar package which will delete the contexts and remove the functionality from the site.

2.1.5 Dependency Relationships

While the usage scenario for package management seems simplistic for basic installations and removals, the nature of software development is that dependency relationships are common between separate components. Especially in the package system where components implement a specific functionality, there may be several requirements for a package installation since it depends on the features in other packages. This dependency is called a required dependency.

To demonstrate the role of required dependencies on the system, suppose the site administrator wishes to install a Photo Album Bookmarks package which requires that the Photo Album package is installed. For the sample site configuration in Figure 2-4, the installation would proceed normally since the required dependency is present. However, if the Photo Album package was not previously installed, the site administrator will be prompted to resolve the dependency and install a compatible version of the Photo Album package either by uploading the package file or retrieving the package from the repository through web services. Again it is expected that system administrators will perform dependency resolution through the packages retrieved from the repository. As with a basic installation procedure, the web services eliminate the need for the site administrator to search for the correct package and obtain a local copy to upload to the package manager.

The workflow of this required dependency resolution process is shown in Figure 2-6, and can be extended for the installation of a more complicated web of required relationships. The specific mechanism for the general case is described in detail in

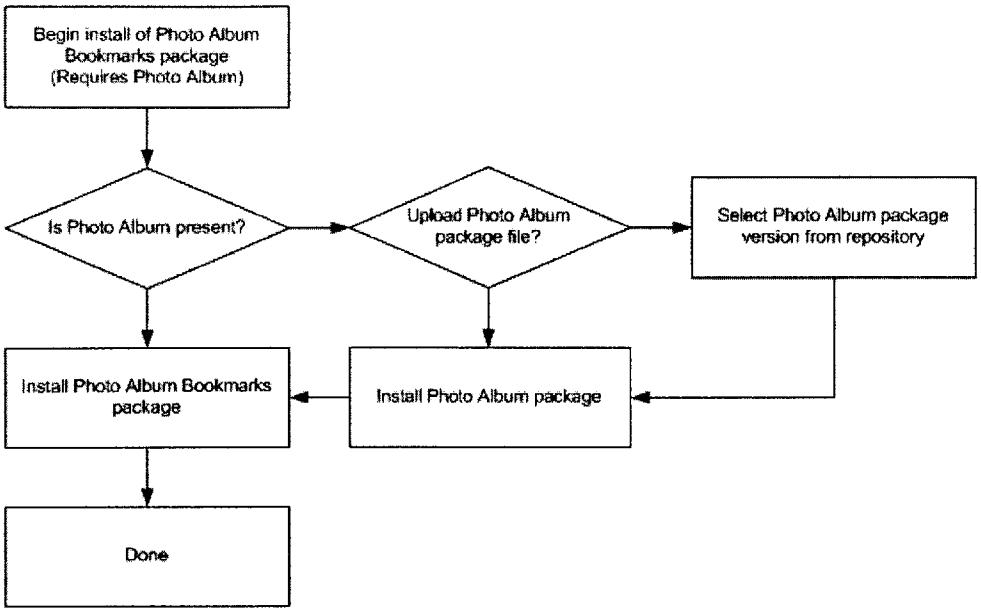

Figure 2-6: Workflow for Required Dependency Resolution during Installation

section 4.2.5. There the dependency resolution implementation is considered in depth since required dependency relationships may result in the installation of multiple packages in one procedure, where the order of commands must be correct for a working system.

The opposite of a required relationship is a conflicting relationship where two conflicting packages cannot both be present on a system at any time. This rule is enforced during the installation and upgrade processes, and displays an error message to the site administrator if any attempts are made to add a package conflicting with any currently installed package. Suppose the site administrator finds a Message Board package which conflicts with the Discussion Forum package because they have conflicting namespaces. An unchecked installation would result in broken pages due to ambiguous naming of namespaces and classes. Therefore, if the site administrator carelessly attempts to install the conflicting Message Board package, the package manager will refuse to install the package and display a message indicating the violated relationship.

2.1.6 Repository Community

As demonstrated in the installation and upgrade processes, the package repository offers useful web services allowing site administrators to easily keep their packages up to date. Furthermore the repository allows dependency resolution so that required components can be easily retrieved and installed. Finally, the repository also allows site administrators to participate in community discussions through their local package management pages so that users and developers of the package management system can share their knowledge with each other.

The package community discussion focuses on particular packages and the site administrator can simply view or submit comments through the package details page. Although the comments are actually stored at the repository, web services make it possible for the site administrator to easily obtain relevant information associated with a package.

One important case where this is useful is the ability to quickly share knowledge about a package, for example, if a bug is found. When a site administrator discovers a bug in a package, he can simply submit a comment describing the problem and any relevant fixes through the details page of the faulty package. Other users of the package management system will also receive the message and be able to make the precautionary fix as well. The package developer, upon receiving the notification of a bug, can also make changes to the package and respond by releasing a new version with the fix. Once the package is placed in to the repository, it becomes available to all the users of the package management system.

2.2 Web Site User

Certain features of the package management system are intended to create a more user friendly experience. Since the package management system is intended for web based applications, particularly community web sites, this section describes how the package system creates a usable environment for a member of the web site.

2.2.1 Portlets

Since packages generally have components intended for display through web pages, they may optionally supply custom controls providing a summarized view of the package. These custom controls known as portlets are intended to contain the most recent or important data associated with a package for placement on portal or other aggregation pages. Portlets have the ability of being contextualized, thus users can customize the layout and content of their personal portals.

2.2.2 Path Resolution

The path resolution mechanism not only serves as a mount point for a package context, but also allows pages to be served at human readable URLs. This feature is an aesthetic benefit for users who type URLs directly in to their browsers, or wish to save readable URL bookmarks.

2.3 Package Development

The package management system is only useful if developers create packages to be used for web sites. Therefore, a package system without benefits for the developer would be unsuccessful since the community would never develop. The process of creating a package is intended to offer many benefits to the package developer in terms of convenience and core functionality so that programmers can exploit the features to expedite their development and release cycles.

Mainly the argument for using a package system is that the modular design is more manageable and flexible than a large monolithic design. The package management system is the framework that allows developers to collaboratively work on individual components rather than forming complicated intricate dependencies in a large system that will inevitably be inadaptable to the variety of needs that web site administrators demand. Here we discuss the more detailed advantages of using the features supported by the package model from the perspective of the developer.

2.3.1 Contexts

Since the package management system understands the concept of package contexts, the framework simply informs the package of the current context. This eliminates the burden on the developer to organize the package contexts or attempt to support multiple installations; instead the developer uses the assigned context to retrieve and apply any configuration settings.

2.3.2 Dependency Relationships

Developers who wish to develop packages for the system will be able to include the functionality in other packages by specifying a required dependency. This prevents developers from reinventing solutions so they can decrease their development time and efforts.

2.3.3 Multiple Database Support

Web site developers often face the difficultly of supporting multiple database platforms since subtle differences in database architectures may result in data models being incompatible across multiple systems. To alleviate this problem, the package manager supports the usage of a library to abstract the database type from the developer.

2.3.4 Package Repository

The benefits of participating in the package repository community are two fold. First the package developer has access to a medium where completed work is quickly and easily distributed to a large audience of users. Since the repository can be accessed by other package management systems through web services, site administrators can immediately browse recently submitted packages and make use of the developer's work.

The second advantage of the package repository is that the wide audience creates a community of site administrators willing to use and test the new functionality.

The site administrators who share feedback will provide useful information to developers about the positive and negative points of the package, as well as possible improvements for future versions.

Chapter 3

Overview

Principles such as modularity and code reusability are not new to software system designs. The concept of a package manager has in fact been applied to other systems to achieve these design goals. Here we discuss the design goals and implementation of our own system, some based on the existing systems involving the dynamic addition and removal of componentized pieces of software, while others are emphasized for the purposes of a web based application. The design goals are also focused on creating a system to exploit the introduction of web services to a package system for the benefit of site administrators and developers.

3.1 Design Goals

The goal of this thesis is to provide a package management system that simplifies usage for both developers and web site administrators. Ease of use through basic operations is an important aspect, but without losing the ability to make packages customizable and flexible in terms of both implementation and functionality. To further enhance the usability aspect of such a system from previous work, user feedback in a community type environment will be used to ensure that developers and administrators can be quickly alerted of recent developments and discoveries related to packages relevant to their usage.

This thesis will incorporate several important ideas from the OpenACS system

[16, 17]; however, this is certainly not a port of an existing system. Although we will borrow several ideas for a local package management system, this thesis will offer ways to improve web site package administration through the use of web services interaction with a central package repository in conjunction with the local system, further described in section 6.3. Following the scenario description in Chapter 2, we can now specify and define the design goals of the local package management system.

3.1.1 Packages

As with the previously described package management system, a package is a modular encapsulation of a single functionality or feature for usage in the system. A calendar system, a message board or a pluggable authentication module are all possible examples of packages. This separation of functionality is beneficial for the web site administrator who can easily manage individual components by installing new packages, removing unused or obsolete packages, and upgrading individual packages to their latest versions. Developers also benefit since writing code to modify or add behavior to a site affects only individual packages and their dependencies.

Since a package may implement a specific feature that should be duplicated on a web site, the design must support the ability to contextualize and mount packages. By this we mean that installing a package simply means that the functionality exists in the system. However, to activate the package at a viewable URL, the site administrator must create a context of the package mounted at a URL mapping through a web based user interface. Several contexts of a package can be made, unless a package specifies that it is a singleton type meaning it allows at most one context.

The design supports package contexts so that web sites can easily duplicate functionality. An example would be the use of message boards on an educational web site where each class would have their own message board for group discussions. In this usage scenario, each class would have a mounted context of a message board, so that the message board package would be installed once on the system, but used in multiple contexts.

A further benefit to this design is that individual contexts may have their own

configuration settings. Each package defines the configurable parameters that are unique to separate contexts. Following the example of multiple message boards for classes on an educational web site, the message board package could perhaps supply configuration parameters for threaded view, access control, background color and other miscellaneous user interface settings. The point here is that individual contexts are not only independent of each other in terms of content and data, but can be contextualized by their configuration settings as well.

3.1.2 Package Manager

The package manager is the mechanism by which packages are organized on the system. A major design goal with respect to the coordination of individual packages and their associated files is to provide a package management system that simplifies the operations of package installation, upgrade, and removal for web site administrators. The primary interface will be a web based user interface, but these operations should be exposed by an API that allows future developers to create other interfaces such as command line tools and windows based applications.

There are several complications with the package operations that must be considered by the package manager. The most important consideration when package installations are modified on the system is the management of their dependency relationships. Since packages are designed to contain specific functionality rather than supplying an entire collection of features, it will be quite common for packages to depend on other packages for certain functionality. This type of relationship is a required dependency where a package cannot be installed if the dependent package is not also included in the system. Similarly, a package that is depended on by another cannot be removed from the system.

The opposite of a required relationship is a conflicting relationship. As the name implies, in this case, conflicting packages may not both be simultaneously present on a system because they have known issues that prevent proper functionality. Conflicting relationships are only relevant during the installation and upgrade operations since package removal will never violate a conflicting relationship.

3.1.3 Package Repository

Existing package management systems have a web repository of packages as well as a newsgroup or discussion board for users to ask questions and post feedback about packages. The design goal of creating a package repository system to facilitate the distribution and use of packages is therefore not a new idea for package management. The new innovation is actually the use of web services so that the centralized package repository hosted at a known location is capable of communication with the package management systems.

Other existing package management systems provide the repository and discussion functionalities on the web, but in disjoint locations making it less trivial to determine where relevant information could be found. Placing a web services API exposing key functionality of the repository allows the package management system a great wealth of functionality to simplify administrative tasks. Package installation and upgrade procedures query the repository for a list of missing required dependencies, allowing administrators to automatically download additional packages. The package manager also displays a list of packages which may be upgraded to a newer version by obtaining a list of the latest packages from the repository.

Another useful application of web services is the sharing of information between all the users of the package management system. The web based user interface of the package management system displays all the comments and feedback about particular packages retrieved from the repository, serving as a discussion board integrated in to the site administrator's view of the local system. Site administrators may also use web services to post feedback and comments so that other users of a package may benefit from the information, or developers can quickly receive constructive criticisms or praise for their work.

The design is meant to support a web interface and supply the functionality described. Beyond the scope of this design, but potentially useful web services in such an application may include a ratings system for packages, information regarding usage statistics of a package, or popular alternatives for particular packages. More detail is

given about possible extension of the web services API in section 7.1.

3.2 Implementation

The implementation of the package management system takes in to account the specified design goals. An important aspect to choosing the implementation details is the desire to make the system easily adaptable for users accustomed to different software in terms of database and development environment. The intention is to create a system with fewer barriers of entry in hopes of adoption by a wide audience.

3.2.1 Microsoft .NET

Since web services play a significant role in the package management system, the development platform of choice was Microsoft .NET. The web pages were created using ASP.NET and all source code is in C#, so the .NET Framework Software Development Kit (SDK) [10] is a required portion of the installation. Development work was primarily done in Microsoft Visual Studio .NET on Windows XP, while serving pages from an Internet Information Services (IIS) web server backed by an SQL Server database. Although it is expected that this is a similar configuration to what most users of the system will use for development or production environments, here we discuss the alternatives for users wishing to use non-Microsoft software.

3.2.2 Platform Independence

The package manager currently requires running on the Microsoft Windows platform mainly because the .NET Framework requires installation on Windows. However, the Microsoft Shared Source Common Language Interface (CLI) Implementation (code named "Rotor") [20] and the Ximian Mono Project [22] both show great potential to allow support for the .NET Framework and thus the package management system on other platforms in the future. At the time of development, the priority was to create a working system for the Windows platform, and consider ports to other platforms

when the Rotor and Mono projects have matured.

3.2.3 Database Support

To avoid locking in users to a Microsoft SQL Server database, all database interaction in the system is abstracted through the iLearn multi-database API which hides the specific database type backing the system. The library offering this support was written by Andrew Grumet and can be seen by obtaining the source code and compiling the documentation. For more information, see section 8.1 for instructions to access the code through CVS.

By using the multi-database package, developers place named queries in XML files segregated by queries that are specific to SQL Server, Oracle or PostgreSQL, or use standard SQL92 syntax. When executing a database command, the multi-database package uses the correct query depending on which database is installed on the system. Through this mechanism, the package manager and other packages can rely on the multi-database code to easily support multiple databases.

3.2.4 Development Software

Rather than forcing users to type long compilation commands using the Microsoft C# compiler bundled with the Microsoft .NET Framework, we require NAnt [12] as the build environment for usability purposes. NAnt is a free .NET build tool allowing the use of scripts conforming to an XML specification to build the source code for the package manager and packages. Through NAnt and the associated scripts, packages include build files that specify the source files, references and other parameters required to properly compile the source.

By separating the build and text editing environments, developers can use their text editors of choice. Thus package developers who wish to use Microsoft Visual Studio .NET as their development environment for editing and building may do so, and those who prefer other text editors and simply use NAnt for compilation.

Chapter 4

Package Management System

Before discussing the package repository and the exposed web services, this chapter describes the design details of the local package management system. This includes the specification of packages, and the workings and features of the package manager.

4.1 Packages

As mentioned earlier, packages provide the functionality to be used in a system organized by the package management system. These modular code components are responsible for implementing specific features and give site administrators the power to easily modify the feature set of their web site by installing, removing, or upgrading packages.

Packages are distributed as single ZIP compressed files conforming to an expected specification set forth by the package manager. Here we discuss the expectations of a package including naming conventions, implementation of the package interface, file system structure and the XML specification file describing the properties of a package release.

4.1.1 Naming Conventions

Because namespaces must be unique to a particular application, we make use of the package's namespace for several conventions. We also set the convention that packages are nested under the `ILearn.Packages` namespace. The distributed package file is named after the namespace and may optionally be followed by the version number for the package separated by a hyphen. For example, the fictitious package `MyPackage` whose namespace would be `ILearn.Packages.MyPackage` could be distributed in a file named `ILearn.Packages.MyPackage.zip` or assuming that the version number is 1.0, `ILearn.Packages.MyPackage-1.0.zip`.

Packages are rooted at subdirectories also named after the namespace. However, because serving web pages from subdirectories whose name contains a period causes problems for IIS, the convention for the root directory is to replace periods with underscores. The sample package would therefore be rooted at a directory named `ILearn_Packages_MyPackage`. Note that the version number is not included in the package root directory.

Finally, it is expected that all the standard files are named after the namespace with the appropriate extensions. These files include the NAnt build file, the Microsoft Visual Studio solution and project files, and the standard package configuration file. Section 4.1.2 describes the placement of files within the root directory, and section 4.1.4 describes the package configuration file.

4.1.2 File System Structure

The standard directory structure format is to separate files based on their function as shown in Figure 4-1. The `lib` subdirectory contains C# code supplying backend functionality. ASP.NET pages and their code-behind files are placed in the `www` subdirectory. The `www` subdirectory also contains any other files associated with user-visible pages such as custom controls and images. The `sql` subdirectory contains the XML files containing named queries to be used for multi-database support described in section 3.2.3.

```
ILearn_Packages_MyPackage\
  |
  + -- lib\
  |      |
  |      + -- MyPackage.cs
  |
  + -- sql\
  |      + -- oracle\
  |      |      |
  |      |      + -- ILearn.Packages.MyPackage.xml
  |      |
  |      + -- postgresql\
  |      |      |
  |      |      + -- ILearn.Packages.MyPackage.xml
  |      |
  |      + -- sql92\
  |      |      |
  |      |      + -- ILearn.Packages.MyPackage.xml
  |      |
  |      + -- sqlserver\
  |             |
  |             + -- ILearn.Packages.MyPackage.xml
  |
  + -- www\
  |      |
  |      + -- default.aspx
  |      |
  |      + -- default.aspx.cs
  |
  + -- ILearn.Packages.MyPackage.build
  |
  + -- ILearn.Packages.MyPackage.config
  |
  + -- ILearn.Packages.MyPackage.csproj
  |
  + -- ILearn.Packages.MyPackage.sln
```

Figure 4-1: Standard Package Directory Structure

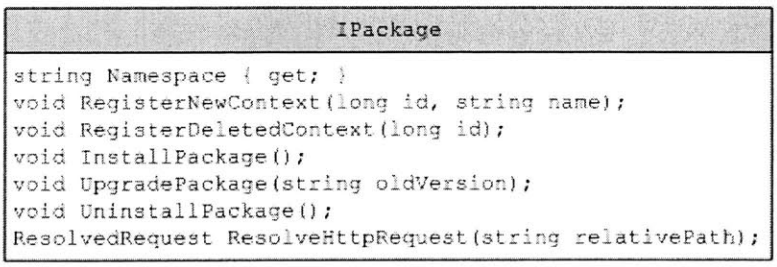

Figure 4-2: `IPackage` Interface Definition

```
namespace ILearn.Packages.MyPackage
{
    public class MyPackage : ILearn.Core.Packages.IPackage
    {
        public string Namespace
        {
            get { return typeof(MyPackage).Namespace; }
        }
        ...
     }
}
```

Figure 4-3: Sample Namespace Property in `IPackage`

The package manager expects to find the configuration and setup files in the root directory of the package. These files include the NAnt build file, solution and project files used with Microsoft Visual Studio, and the package configuration file.

4.1.3 `IPackage` Interface

For a package to be managed by the package management system, the package must adhere to the `IPackage` interface definition by supplying a class that implements `IPackage` from the `ILearn.Core.Packages` namespace, shown in Figure 4-2. The most basic method a package must implement is the property to return the namespace of the package by using a hard coded string or supplying an implementation based on the sample code in Figure 4-3.

The installation and removal methods provide an opportunity for package implementations to perform custom actions during the installation and removal phases of

a package. The most common usage of these methods is the creation and deletion of database tables used specifically by the package. `IPackage` also requires an implementation of the upgrade method which accepts the old version string being upgraded. The purpose of this method is to make any changes to the system necessary for the upgrade. For example, an upgrade may require the creation of new database tables, the removal of old tables or copying data between tables. This allows the package to supply a customized method to preserve data that should be maintained in an upgrade process.

Another purpose of the `IPackage` definition is to perform any custom actions during the creation and deletion of the package contexts. During the creation or deletion of a package context, the package manager will register a new or deleted context supplying the context identifier and the name where necessary. Note that the package implementation does not have to create or remove the package context settings since those values are maintained by the package manager. For more detail about package contexts and context settings, refer to section 4.2.7. The intentions of these methods were to allow the package implementation to initialize any context data, or remove generated data as a result of context creation and usage.

Finally the package must be able to resolve HTTP requests by implementing the request resolution method. Since the functionality of the site is componentized by the package management system, the `ILearn.Core.PathResolution` namespace includes an implementation of `System.Web.IHttpModule` to handle the `BeginRequest` event to resolve requests for server objects. During the handling of this event, the implementation of `IHttpModule` calls the custom request resolution method of the requested package, supplying the relative path of the request. Refer to section 4.2.8 for more information about path resolution in the package management system.

Because it is expected that most packages intend on resolving the request by returning the corresponding page from the `www` subdirectory according to the standard directory structure as specified in section 4.1.2, the `ILearn.Core.PathResolution` namespace supplies an `AbstractRequestResolver` class supporting this default behavior. Packages adhering to the naming convention of placing displayable files under

```
namespace ILearn.Packages.MyPackage
{
    public class MyPackage : ILearn.Core.Packages.IPackage
    {
        public ResolvedRequest ResolveHttpRequest(string relativePath)
        {
            AbstractRequestResolver arr = new AbstractRequestResolver(
                this.Namespace
                );
            return arr.ResolveHttpRequest(relativePath);
        }
        ...
     }
}
```

Figure 4-4: Sample Request Resolution in `IPackage`

the `www` subdirectory can implement the custom request resolver following the sample implementation in Figure 4-4. Of course packages are free to use their own custom implementations if their request resolution scheme is different from the default behavior.

4.1.4 Package Configuration File

Each package must supply a package configuration file named after the package namespace. The configuration file is an XML file used to describe the package properties such as name, description, version, contact, information and type. The configuration file also defines the dependency relationships the package has (conflicts and requirements), describes the portlets supplied with this package, and lists the configuration settings to be managed by the package manager. Finally, the package configuration lists files and references for compilation during installation, or optionally specifies a NAnt build file or Microsoft Visual Studio solution file to use for compilation. Appendix A provides the detailed specification of the XML configuration file.

4.1.5 Versioning

Rather than force each package in to a standard versioning scheme, the package management system allows each package to specify its version number as a free form string. However, this imposed the difficulty of attempting to compare version strings within packages. For example, it would be difficult for the package manager to programmatically determine the relative recentness of packages with the versions "1.1", "1.1.6", "1.02" and "1.1-Beta" which are all legal version strings.

One solution was to use the release date in the package specification, but this scheme would prevent package developers from releasing versions out of order. To solve this problem, each package is required to specify a serial which is an integer used by the package manager to compare versions where a larger serial indicates a newer package. The serial also allows a more robust mechanism of specifying package dependency versions where asterisks can be used as wildcard characters ("1.*" would specify any package version string starting with "1.") or comparison operators can be used with the serial (">10" would specify any package with a serial greater than 10).

4.2 Package Manager

The package manager refers to the user interface and backend functionality available to web site administrators that organizes the packages on the local system. The operations supported by the package manager for organizing packages on the system are installing, removing and upgrading packages. These operations also include a dependency resolution and checking mechanism to verify that package operations will not cause the system to be in a state where package relationships are unmet.

To make the package system more flexible, the package manager also handles the notion of package contexts and their configurable settings. Finally, since the package manager is geared towards usage for a community web site, packages may optionally include portlets. Portlets are custom controls providing a summarized view of a package context for the purpose of placement on a portal or aggregation page.

This section begins with an overview of the package implementation with a data model and class structure, and then continues to describe each of the operations and features of the package management system. Finally, the section ends with a description of the user interface for package management.

4.2.1 Data Model

Figure 4-5 shows the data model used by the package management system. The `core_metadata` table is part of the iLearn core object system written by Andrew Grumet, and for the purpose of this data model is used to provide unique primary keys for the `package_properties`, `package_portlet` and `package_context` tables. When a package is added to the system through installation, appropriate entries are inserted in each of the relevant tables to store the specified configuration of a package. The creation of a package context inserts a row in to the `package_context` table, and also inserts the appropriate context configuration setting values in to the `package_context_setting` table. To uninstall a package, the package manager executes these commands in the reverse order to remove the database entries and ultimately the package from the system.

4.2.2 Class Structure

The main functionality of the package manager is encapsulated in the `PackageManager` class of the `ILearn.Core.Packages` namespace. As shown in Figure 4-6, the package manager contains public static methods intended to be used by the web based interface for the package management operations. Since these methods are public, other interfaces can be developed using the package management functionality. For example, a console based application or a Windows forms application could be written to allow site administrators the ability to organize packages without using a web based interface. The web based interface is explained in terms of an interaction scenario in section 2.1 and in terms of workflow in section 4.2.10.

As shown in the data model in section 4.2.1, there are several aspects of an installed

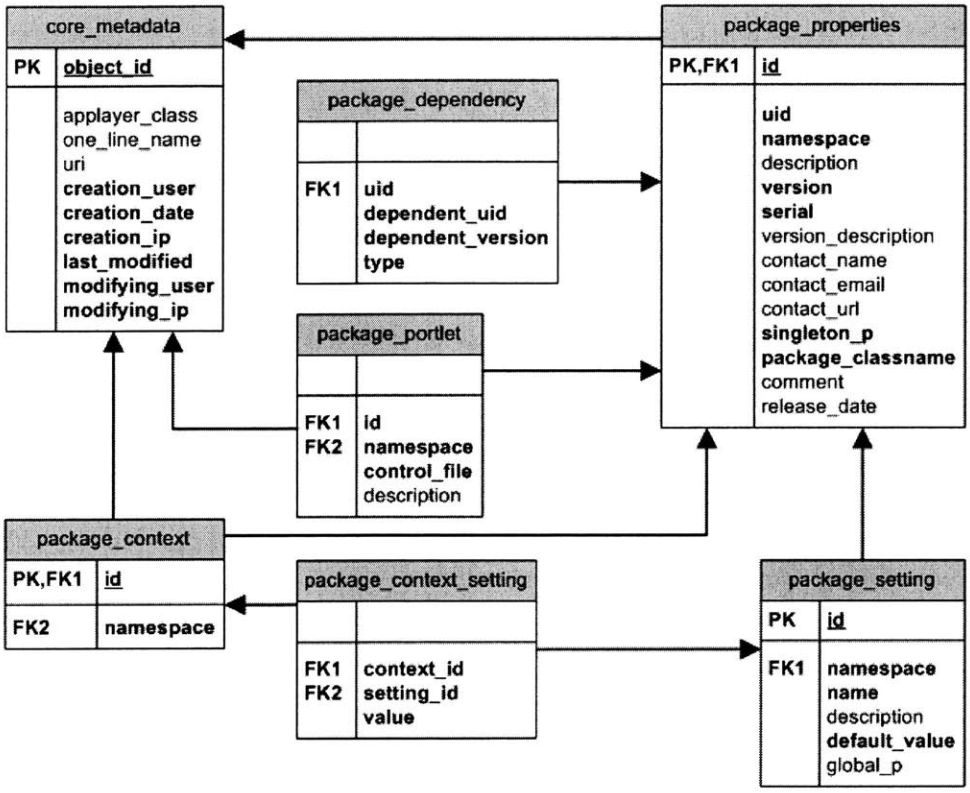

Figure 4-5: Package Management System Data Model

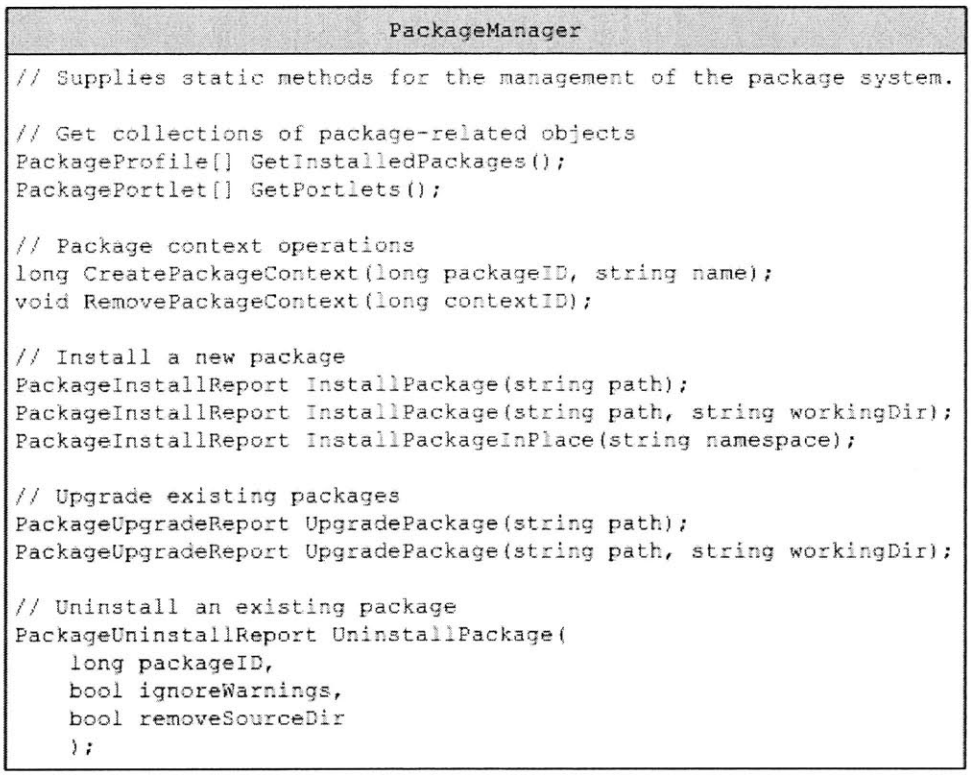

Figure 4-6: `PackageManager` Class Diagram

package to represent in the system. There are the basic properties describing the package, as well as parameters that describe the behavior of the package such as dependencies, portlets and configuration settings. Furthermore, an installed package may have had several contexts created, each with unique settings. Figure 4-7 depicts these individual aspects of the package representation, the relevant properties of each component, and the `PackageProfile` class which aggregates all the classes to form a complete representation of the package in the system. For performance purposes, the `PackageProfile` uses a caching mechanism to partially retrieve the data from the database, and uses in memory values where appropriate.

The final component of the class structure is the `IReport` interface which is used by the package manager to return the results of an operation. Figure 4-8 shows the `IReport` members, as well as the `IDispatchReport` interface which is intended to add methods and properties expected by reports generated for dispatch operations such as installations and upgrades. Not included in the class diagram for `IReport` and

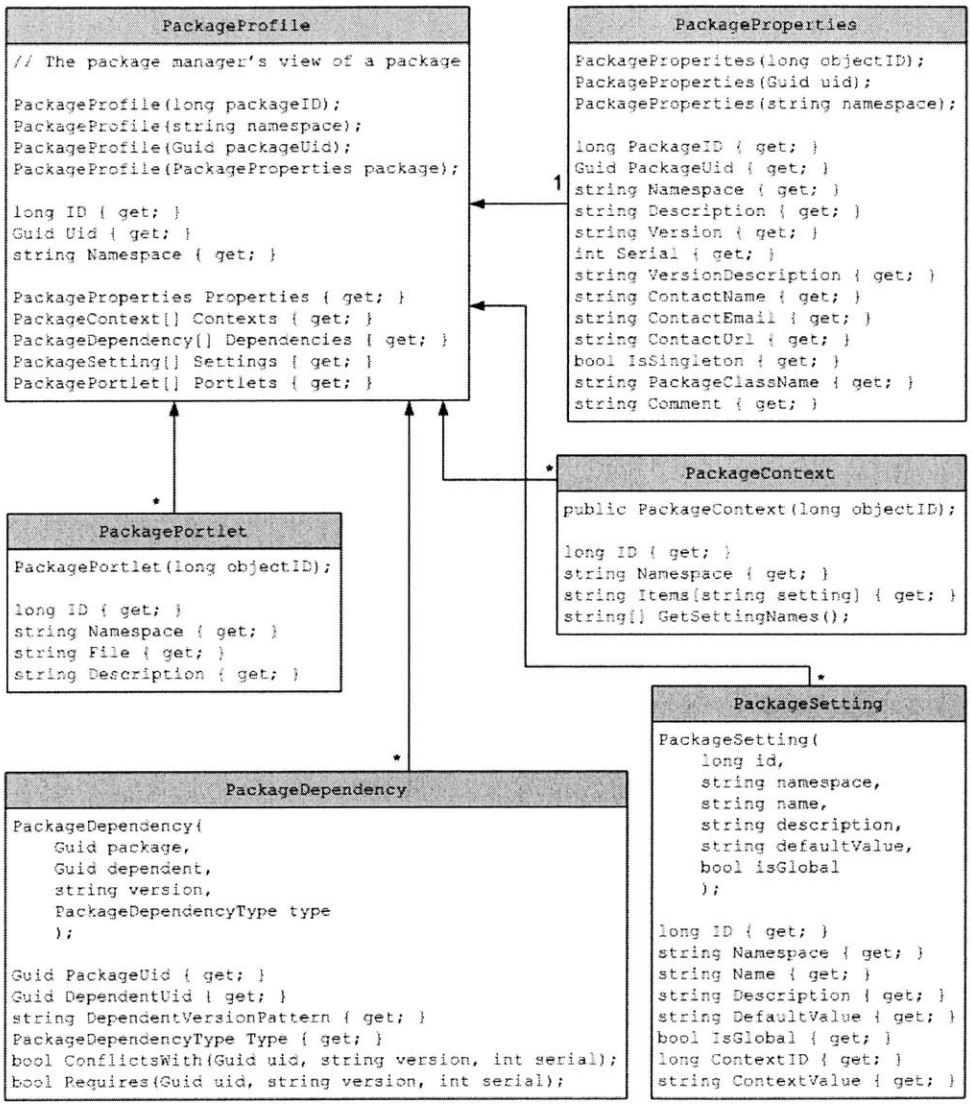

Figure 4-7: `PackageProfile` Class Dependency Diagram

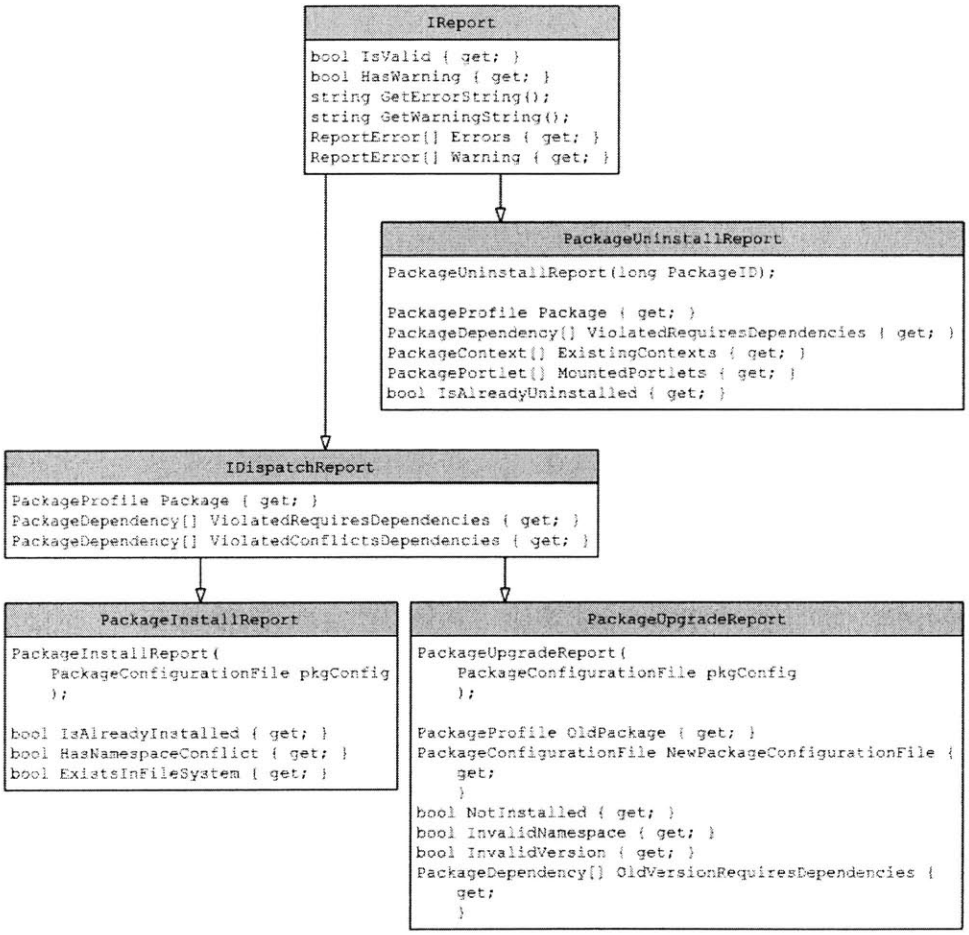

Figure 4-8: `IReport` Interface Definition

implementing classes is the definition for `ReportError` which is simply an enumeration of possible report errors and omitted for brevity. The implementations of `IReport` are `PackageInstallReport`, `PackageUpgradeReport` and `PackageUninstallReport` are returned for the appropriate package manager operations.

4.2.3 Installation

The installation procedure is illustrated in Figure 4-9, showing a flowchart of the package manager's behavior and order of operations. The intent of the package install operation is to fully complete the package installation, or make no changes to the file system and database. Therefore to accomplish this, the package manager simply undoes any generation of files at any signs of error.

`PackageInstallReport` is a safeguard against many potential errors in the installation since it makes system wide checks to verify that the package can be properly added to the system. If the report predicts that an error will occur, the installation procedure simply returns a report with flagged errors without creating new files on the system. Possible errors are that the package is already installed, the namespace is already in use, or the expected location on the file system already exists and cannot be overwritten. `PackageInstallReport` also detects dependency violations, such as installing a package that conflicts with an existing package, or installing a package while a required dependency is missing from the system.

Assuming that no errors are found, the installation operation proceeds by decompressing the package file, compiling the sources of the package, executing the installation method implemented by the package (in the class definition of `IPackage`), then finally appropriately updating the configuration in the database.

4.2.4 Upgrade

The upgrade procedure is similar to the installation procedure with the exception of a few key differences. The upgrade process follows the principle of fully upgrading a package or failing without any modifications to the system. However, to undo

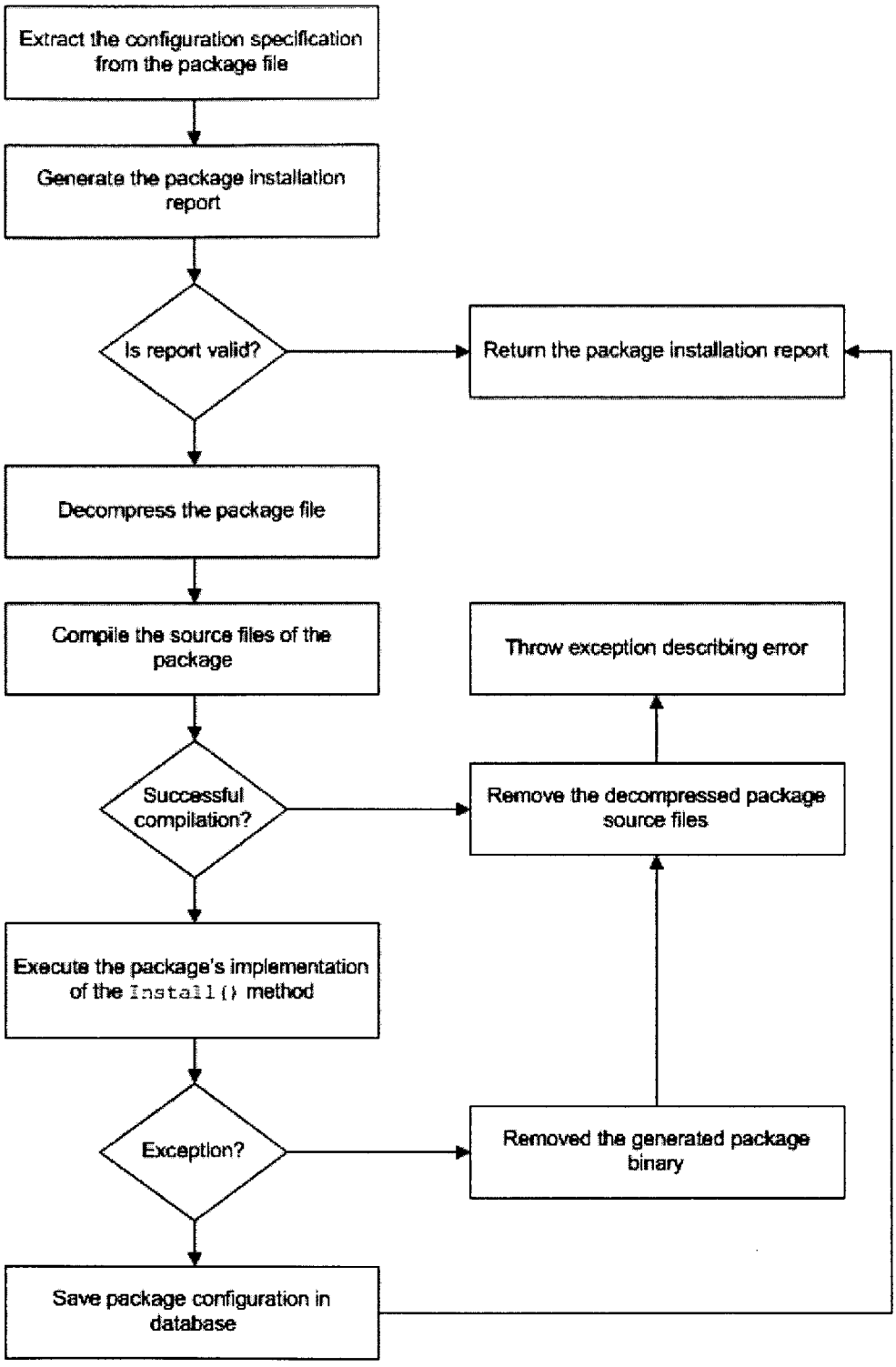

Figure 4-9: Package Manager Installation Procedure

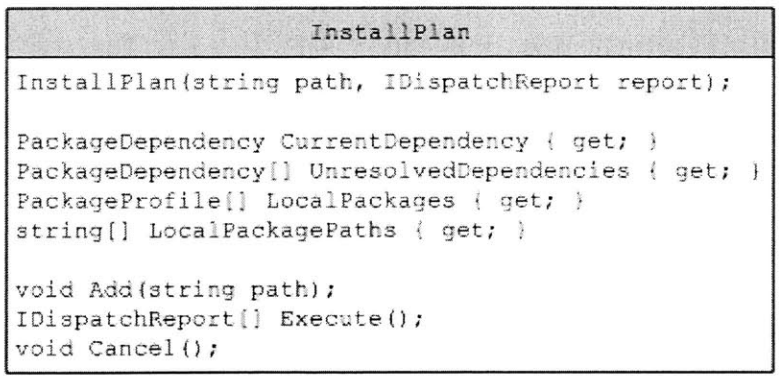

Figure 4-10: `InstallPlan` Class Diagram

the operations of a failed upgrade, the package manager must backup the source and binary files of the previous version to revert in case of errors. Other than the additional backup of files, the upgrade procedure uses the same mechanism of verifying upgrades with a `PackageUpgradeReport` predicting potential flaws.

A non-trivial complication in the final step of upgrading is when the database must be modified to the new configuration of the package. This is accomplished by updating new properties in the `package_properties` table and replacing all the relevant entries in the `package_dependency` and `package_portlet` tables. Finally to maintain the upgrades for package contexts and configuration settings, the package manager removes obsolete entries from the `package_setting` table, appropriately updates any existing settings that are changed in the new version, and inserts new entries while setting the context values in the `package_context_setting` table to the default values.

4.2.5 Dependency Resolution

A complication with package installation and upgrades that should be discussed in further detail is the case where the operation cannot be completed because of missing required dependencies. If this is the only error with the install or upgrade, the package manager is equipped with a mechanism to resolve the required dependencies through the `InstallPlan` class shown in Figure 4-10.

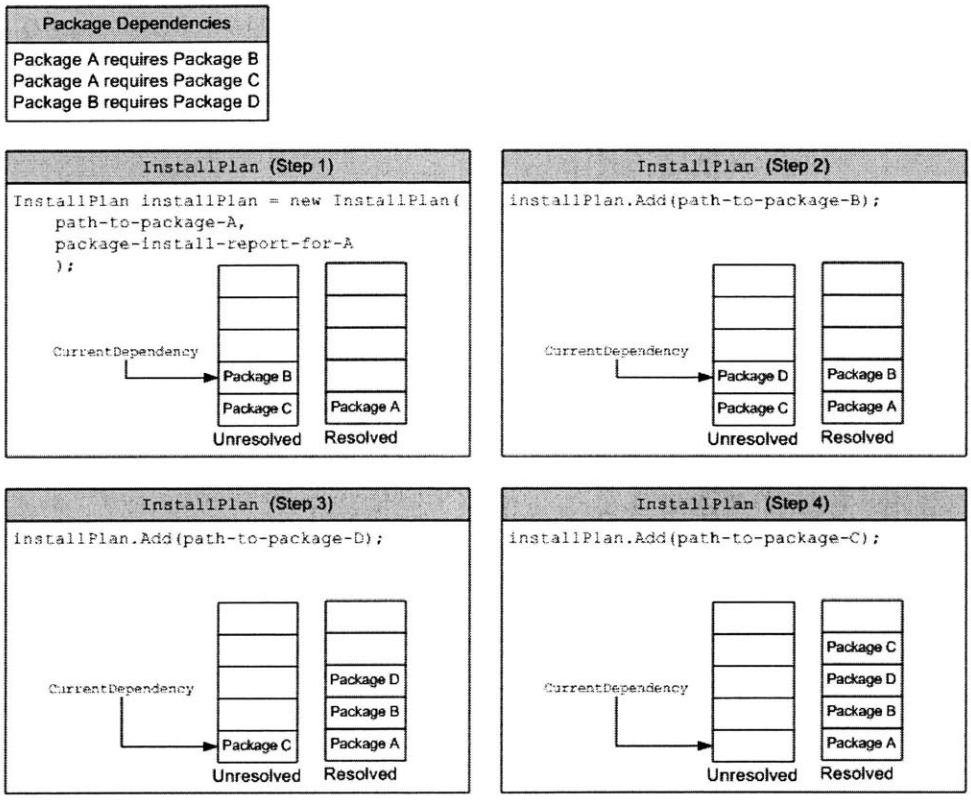

Figure 4-11: Dependency Resolution for Multiple Package Installations

The `InstallPlan` class is constructed with a package file and the associated report for installation or upgrade. The scenario described in section 2.1.5 was a basic example of dependency resolution where the installation of a Photo Album Bookmarks package required the Photo Album package in a two-step installation process. However, as the resolution process illustrated in Figure 4-11 indicates, installations with multiple steps can involve significantly more complication which `InstallPlan` must be equipped to handle.

We expand our example by considering the case where package A is to be installed, but it requires packages B and C which are not found in the system. By adding paths to the required packages the `InstallPlan` will generate an ordering of packages to install so that ultimately package A can be flawlessly installed. This idea is similar to directed graph of procedure calls used in the design for the Tcl Content-Derived Name (Tcdn) package system [11].

Continuing the example, the `InstallPlan` therefore has package A in the list of local packages, and packages B and C are listed as unresolved dependencies. Package B is the current dependency this `InstallPlan` is attempting to resolve. The easiest thing to do is to repeatedly add a path to a package file satisfying the current dependency, used to generate the order of package installations. However, what if package B also cannot be installed because it requires yet another package D?

This is problem is solved by generating the installation plan with two stacks; one stack for the package paths is used to resolve dependencies, and another stack is used for unresolved dependencies. As packages are added to the system, the algorithm is to push the package on to the resolved package stack and to pop the dependency from the unresolved stack. Any new required dependencies are pushed on to the unresolved stack. The current dependency is a peek of the top element in the unresolved dependency stack, so that a completely resolved `InstallPlan` will have an empty unresolved stack.

The completed plan is executed by popping and installing or upgrading the packages from the resolved stack. In our example, the order of installation is package C, D, B, then A. Through this mechanism, each package is installed on the system at

a point where all of its required dependencies are present. Packages C and D are installed first since they have no dependencies, and are required by other packages. The installation of package B is successful because its dependency (package D) was just installed. Finally, package A is successfully installed on the system since each of its dependencies (packages B and C) are installed.

4.2.6 Uninstall

As with package installation and upgrade, removing a package also requires the use of an `IReport` implementation. One difference with the `PackageUninstallReport` is the use of warnings, which are flagged if a package to be uninstalled either has existing contexts or mounted portlets. The uninstall procedure accepts an argument to indicate if these warnings should be ignored, in which case portlets and context are deleted. Errors caught by the `PackageUninstallReport`, which can not be ignored, include a package not found error, and violated dependencies where the package to be removed is required by another package.

Once these issues are handled, removing a package is a straight-forward process. The package implementation of the removal method is invoked, the binary and source files are removed from the system, and the relevant database entries are deleted.

4.2.7 Contexts and Settings

Once a package is installed on the system, the site administrator must create a package context and mount it at a viewable URL. This design allows the functionality of a package to be replicated for several usage contexts so that multiple users can customize their individual configuration settings. The procedure for creating a context inserts the appropriate values in to the `package_context` and `package_context_setting` tables by assigning default values for the context settings. Finally the package manager registers the new package context with the package implementation. Removing a package context simply does the reverse of the creation procedure.

The notion of package contexts is a powerful feature to benefit the site admin-

istrators as well as the developers. Imagine an educational web site where classes have message boards for group discussions. In several architectures, the students of each class are forced to participate on a shared message board context by starting new discussion threads for relevant topics in their classes. This has the disadvantage of grouping several unrelated contexts in to one system, making the usability more difficult than it needs to be. Alternatively in the package management system, each participating class of the framework can have an individual context of the message board designed to carry out their discussions. No longer are the message board users required to be overloaded with irrelevant posts in the case of having one large message board shared by the entire community.

Furthermore, an added benefit is that configurations can be applied at the level of classes for each of the contexts. Functional or user interface changes can be customized through context settings and applied for individual classes so that the message boards are uniquely tailored to the purposes of each class. This would allow for example, one class to have a moderated message board with threaded views, while another class uses a public message board with a flat view. More trivial customizations such as colors and fonts can add an additional sense of user friendliness to a package context as well.

4.2.8 Path Resolution

While it might seem burdensome for package developers to create the functionality to allow package contexts and settings, the package management API supports this feature quite elegantly with help from functionality in the `ILearn.Core.PathResolution` namespace written by Andrew Grumet. Because of the componentization of the package management system, the path resolution model in the system implements the `System.Web.IHttpModule` interface so that the `BeginRequest` event can be appropriately handled to resolve requests for server objects.

Since mounted package contexts are registered by the site map, the path resolution module is able to use the components of the URL to determine the requested object and the intended context. The path resolution module then rewrites the path to the

requested object, and places the context identifier in to the current session. From the package developer's perspective, each page load will have a context identifier stored in the session which is used to construct a `PackageContext` object corresponding to the requested context. As shown in the class diagram in Figure 4-7, `PackageContext` supports a simple interface for retrieving configuration settings values. Demonstrating the simplicity of using the available API is shown in Figure 4-12 which is the source code of a page that would customize the colors on a calendar web control based on the context settings.

4.2.9 Portlets

Portlets are custom controls providing a summarized view of a package context for the purpose of placement on a portal or aggregation page. This feature is a general improvement to the usability design since it allows the package management system to be better suited for large portal web sites. A portlet is specified by a name, description and control file that displays the contents. The usage model is that portlets are registered with the system by insertion in to the `package_portlet` table during installation. Users of the system are presented with a portal customization interface to select which portlets they wish to display, or mount, on their portal pages. A customized portal page is often the start page for a user, allowing him or her to view summaries of any relevant portions of the entire site, and allowing quick navigation to the items of interest.

To implement a standard portlet in the system, the package must supply a user control that inherits from the `PortletControl` abstract class found in the iLearn portals framework written by Tracy Adams. Inheritance from `PortletControl` requires portlets to supply properties for the context identifier displayed in the portlet and the site node where the portlet's package context is mounted. It is also required for the portlet to have properties that allow customized header and footer controls to be set. This gives the parent portal the ability to dynamically customize the look of each portlet on a page, making it possible to conform the user interface of the portal.

```
using System;
using System.Web;
using System.Drawing;
using ILearn.Core.Packages;

namespace ILearn.Packages.Calendar
{
    public class _default : System.Web.UI.Page
    {
        // The calendar control displayed on the page
        protected System.Web.UI.WebControls.Calendar calendar;

        private void Page_Load(object sender, EventArgs e)
        {
            // Get the package context for this request

            long contextID = Convert.ToInt64(
                HttpContext.Current.Items["PackageContextID"]
                );
            PackageContext pkgContext = new PackageContext(contextID);

            // Set the calendar colors based on context configurations

            calendar.BackColor = Color.FromName(
                pkgContext["Background Color"]
                );
            calendar.BorderColor = Color.FromName(
                pkgContext["Border Color"]
                );
            calendar.ForeColor = Color.FromName(
                pkgContext["Foreground Color"]
                );
        }
    }
}
```

Figure 4-12: Sample Code-Behind Page Demonstrating Contexts

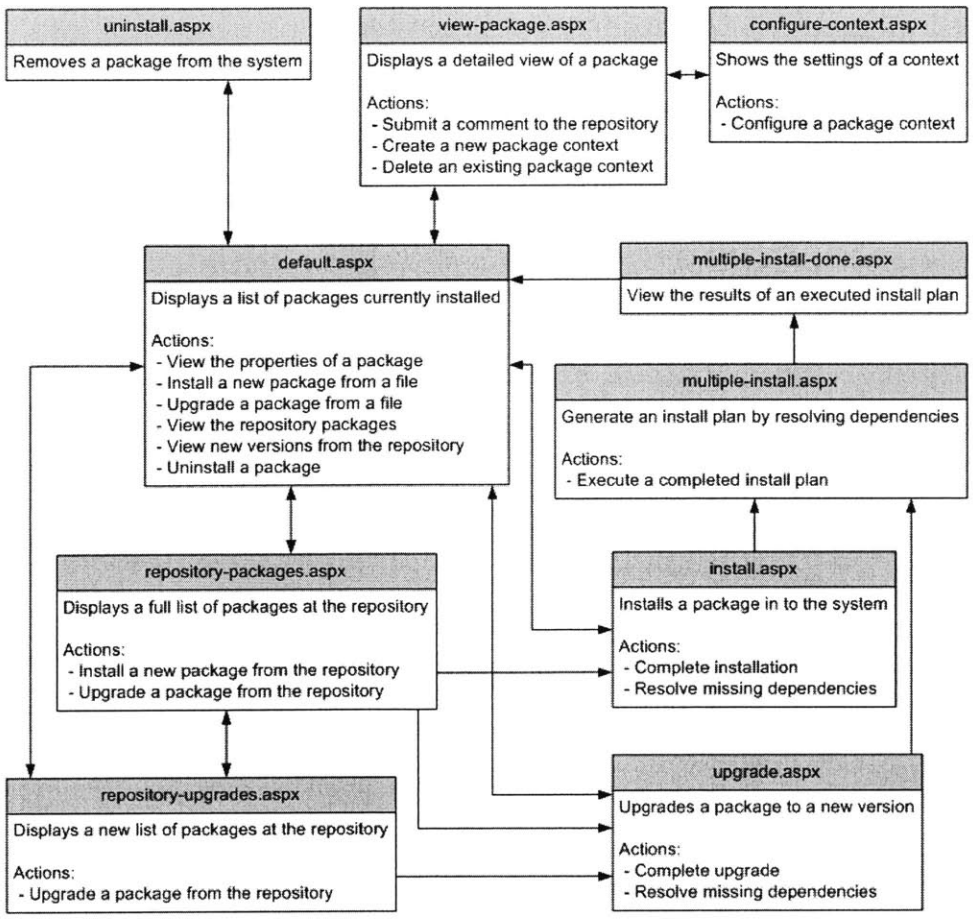

Figure 4-13: Work Flow of the User Interface of the Package Manager

4.2.10 Web Based User Interface

Figure 4-13 illustrates the work flow of the web based user interface for managing and organizing packages. Staring at the default page, the site administrator views a list of package summaries installed on the system. Possible actions include viewing the details of an installed package, installing or upgrading a package by uploading a file, uninstalling a package, or viewing either a full list of repository packages or a shortened list of packages that have newer versions. Actions involving the repository are discussed in further detail in sections 5.2 and 5.3, describing the implementation specifics behind the web service usage scenarios from section 2.1.

Chapter 5

Package Repository

The package repository is a central location for developers to submit new packages for usage on other systems, or for site administrators to find and download new components. Implementing a package repository as a central location for locating packages is not a new idea and is commonly found in several existing package management systems. What distinguishes the iLearn package repository from others is the layer of web services built on top of the repository system to allow package management systems to consume the public API.

This chapter begins by discussing the basic functionality of the repository as a web site separate from the package management system. The web services API will then be explored in depth, with an explanation to justify the specification of the API. Finally, the chapter will end with a description of how the API is used with the package manager functionality to improve the usability for web site administrators.

5.1 Functionality

The functionality of the package repository is two fold. The repository is designed to act as an independent site separate from the package management system, much like the FreshPorts [4] repository for the FreeBSD system. Users are thus given the option of participating on the repository for either posting or obtaining packages without actually maintaining a package management system of their own. This aspect of the

design gives the package repository a look and feel similar to that of a community web site. The other aspect of the functionality is the web services allowing public consumption of the core features found in the package repository.

5.1.1 Community Site

The web site for the package repository supports the obvious requirements of including a basic user registration and authentication system (by e-mail and password), forms for submitting new packages, and pages allowing users to view package details and obtain the package file. To harvest the knowledge of experienced users, the repository site also allows registered users to submit comments and feedback concerning packages to share information, usage experience, answer questions about a package, and generally interact with other users.

While many existing package systems have message forums or mailing lists for user discussions, the iLearn system purposely avoided creating several disjoint web communities relating to the same topic. It is much more useful for users to consult a single source for information, rather than searching through a variety of different places. For example, FreshPorts, the previously mentioned FreeBSD ports repository, provides a good interface for finding and obtaining ports, but users interested in downloading ports receive no user feedback. Instead, users are left to do their own research on the web or perhaps post a question in the sol.lists.freebsd.ports newsgroup. Rather than splitting the information at two separate communities, the iLearn system allows the package details page to include user feedback that may provide useful information influencing a user to either download the package or find a better alternative.

5.1.2 Web Services

The features of the package repository as a community site are useful as web services as well. The main purpose of the web services is to provide information for package management systems that could facilitate site administration. Therefore, information such as posted comments, package listings, newest packages, and possible dependency

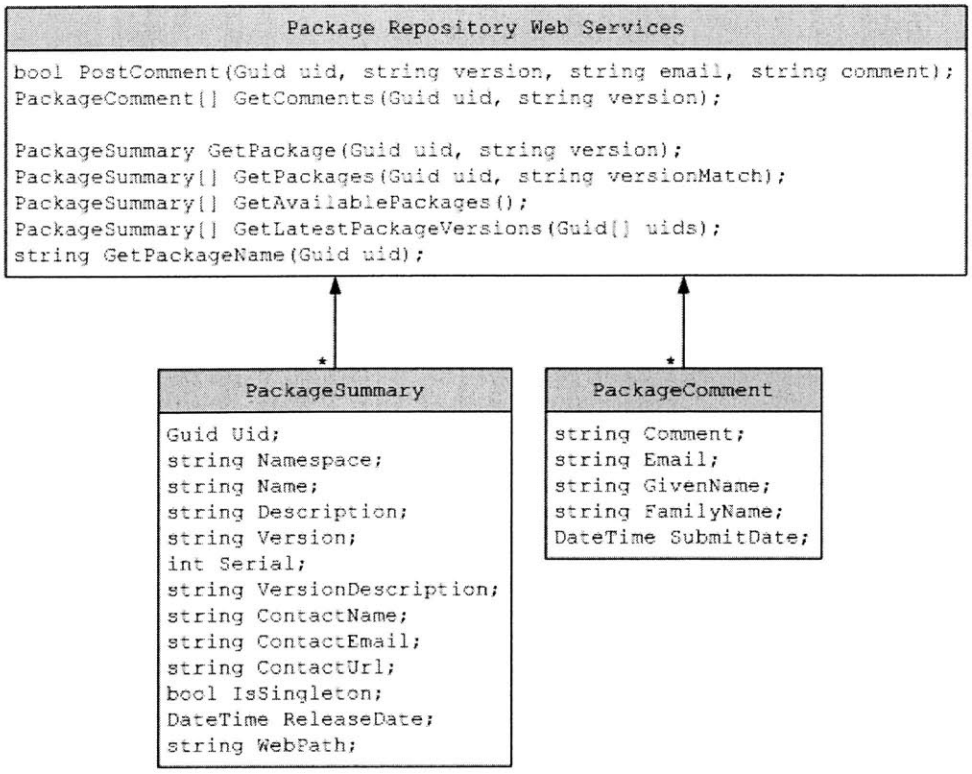

Figure 5-1: Package Repository Web Services

resolutions would provide the greatest benefit for site administrators. Section 5.2 details the web services API supported by the package repository, and section 5.3 explains how the API is used to benefit the package management system.

5.2 Web Services API

To satisfy the goals of the web services for the package repository, Figure 5-1 shows the API for public consumption. First the web services for package comments are rather straight forward; the repository simply allows comments to be read or posted. Notice that packages are identified by globally unique identifiers since the entire collection of packages may potentially have overlapping names or namespaces. For any given package, there may be multiple versions; therefore, the list of packages has a unique constraint on the globally unique identifier and version pair. These parameters are used to uniquely identify which package the consumer is attempting to retrieve or

post comments for, as shown in the method declarations for retrieving and posting comments.

The same mechanism is used to identify a package to obtain a summary of the package, but note that the version number is not required when getting the package name because package names are consistent across each version. To support web service consumers wishing to browse the entire list of packages, the repository exposes a method to return a summary of every known package. Although sending every summary in XML may be rather time consuming as the repository grows, future work described in section 7.1.1 explains how this problem will be solved.

Because getting each package may consume too many resources, the repository also exposes utility methods that return a subset of the available packages by filtering unwanted results, instead of requiring the web service consumers to retrieve all the packages before applying filters locally. One such method accepts a string indicating the desired version or versions of a package to retrieve. Also, instead of returning every package version of the specified globally unique identifiers, the repository exposes a method that returns only the latest versions. The intended uses of these methods are described in section 5.3.

5.3 Package Management Integration

Given the web services API at the package repository in section 5.2 and a description of the package manager user interface in section 4.2.10, we can now explore the intended usage of the web methods exposed by the package repository with the package manager. This section describes the pages in the package manager web interface which use the repository services, but first we mention two notes about performance optimization.

As the number of packages grows, the amount of XML data returned by the repository increases as well, potentially causing the latency of the web method to reach intolerable levels. To solve this problem, the package manager will cache data from the repository in the local database and simply ask for updates at periodic

intervals, rather than receiving fresh data at every page hit. This idea is explained in further detail in section 7.1.1.

Another performance issue deals with a timed out request sent to the repository. Since a page in the package manager cannot be rendered without the information from the web services, an unreachable repository causing a time out would result in several pages to have intolerable delays before rendering. This problem will be better solved when the cached repository data can be used to render a page. However, before the repository data caching mechanism is in place, we employ a simpler solution by disabling the use of repository services when a time out occurs. Future calls to the repository are then ignored to improve performance until the services are manually re-enabled. This protects the system against repeatedly waiting for a web service response on each page render when the repository is unreachable.

5.3.1 Feedback and Community

As part of the design goal to make use of information from knowledgeable users of the package management system, the package manager renders the package details page with the comments retrieved from the repository. Furthermore, to obtain information from site administrators who have working experience with the packages, the interface also allows site administrators to post comments through web services. This will allow site administrators to use the functionality of the repository through their own package manager interfaces, without visiting two different web locations.

5.3.2 Obtaining Packages

Packages are obtained from the repository through the package manager interface can be done programmatically by invoking an installation or upgrade procedure, or simply downloaded since the `PackageSummary` class contains a property specifying the web path (see Figure 5-1), allowing the package manger to link the appropriate file. Programmatic fetching of a package simply opens a request to the web path and writes the file to a local stream, then invokes the install or upgrade procedure on the

local package file.

5.3.3 Dependency Resolution

Packages are also obtained during dependency resolution when multiple installations are required. In this situation, a required dependency must be resolved for an installation to complete. The package manager uses the repository web method to specify the required package and the version match string used to indicate which versions of the package are required for installation. The package manager then displays a list of possible packages to obtain for the purpose of resolving the required dependency, or offers the option of uploading a new package file to satisfy the requirement.

Chapter 6

Comparison with Related Work

The idea of a package manager and the associated principles of modularity and code reusability have been applied to many previous systems. Operating systems such as Red Hat Linux [1, 21], FreeBSD [13, 14, 15] and Debian [3] offer their own implementations of package management systems for organizing installed components on a computer. These principles have even been applied to EMACS [19], an extensible, customizable display-editor developed by Richard Stallman. This concept is less explored in the realm of web site administration and development, with OpenACS [16, 17] being the most successful implementation of a package management system for web based applications.

6.1 Operating Systems

The design of the iLearn package management system is based on the principles used by the Red Hat Linux, FreeBSD and Debian package management systems. There are few appreciable differences between the package systems in these designs so a general comparison will suffice. The general principles shared by the all of these systems are modularity, code reusability, and the desire to avoid monolithic designs. In terms of feature sets, installation and removal were obvious necessities, and the upgrade operation was included to accommodate the need for changing software while preserving the data associated with each package. These ideas are present in the

iLearn system and the operating systems.

Several details involved in package organization for the operating systems were intentionally not included since creating many intricate features would go against the design goal to make the management process straight-forward for site administrators. Considering the installation procedure, operating systems support multiple installation parameters allowing the user to specify a variety of options which would be unnecessary for a web application. For example, the RPM allows forced installations which will ignore conflicts, or can optionally skip dependency checking. Although modifying the iLearn code to support this option is not difficult, it adds a complication to the system which may eventually result in an inconsistent state. Compiling a package while dependencies are unmet would also potentially result in a variety of errors including namespace and class redefinitions, missing references, and a corrupted data model.

Another installation option the design opted to ignore is the ability to set the installation target. Specifying the installation target is appropriate in an operating system, but for a web application created to simplify the administrative tasks and fully function without any knowledge of the underlying file system, allowing this detailed parameter for installation would also be an unwanted complexity. The interface was specifically designed in a way that the site administrator desiring a certain feature would only have to find the appropriate package, upload the file through the web page or retrieve it from the repository, and the installation would complete. These arguments about installation are also applicable to the upgrade procedure which is meant to be a simple process as well.

6.1.1 Debian Relationships

The Debian Package System supports additional weaker relationships such as suggested or recommended relationships. While additional relationships were considered in the system, the benefits did not outweigh the costs. Supporting optional dependencies would create complications since the installation of a package would require compilation with an optional reference to the suggested dependency. Should the sug-

gested dependency package be installed at a later date, the original package would also require recompilation because the suggested reference would then become present.

However, the most important factor in not supporting suggested relationships is the burden it would place on the developer. Consider installing a package with a missing suggested relationship where package A suggests package B because of the recommended use of a function call in package B. Compiling package A without a reference to suggested package B (when package B is missing) would result in a compile time error because the function would be an undefined symbol. For the package developer to work around this and allow suggested packages to be missing, the released code would have to include many pre-compiler directives indicating all the possible arrangements of missing or present suggested relationships wherever any suggested functionality is used. If and when package B is installed, the reference would be present and package A would have to be recompiled but with a different pre-compiler directive. While this recompilation does not seem burdensome in a simple case, multiple suggested dependencies could cause an installation procedure to require several compilations of packages.

6.2 EMACS

EMACS is a "real-time display editor which can be extended by the user while it is running" [19]. EMACS is developed in a modular fashion so that users can add new or modify existing editing commands while in the process of editing. Through the use of collections of function names known as libraries, EMACS loads function definitions in to a dispatch table allowing the command dispatcher to call functions by name or by single key mappings. The iLearn package management system attempts to draw a parallel to this design in the web application world by allowing packages to interact through public methods. However, instead of maintaining a table of function calls, it is more appropriate in the system to maintain a listing of namespace references because of the differences in LISP and C#. While LISP is an interpreted language allowing EMACS library functions to be dispatched by name, the package management system

must compile the C# source code with references to binary files for the public methods to be used by other packages. This is also the reason binary files are named after namespaces, to prevent an ambiguous binary file name to namespace correspondence.

This design is more appropriate for a web based application because of the intended usage. In EMACS, a user attempting to modify an editing command but attempts to dispatch a non-existent function will receive an error message when the particular command is used. The analogous mistake in the package management system would result in errors on publicly viewable web pages if a command dispatcher mechanism was used. Therefore, it is more appropriate to prevent the improper reference during the dependency checking phase of installation, and verify that the function is valid during the compilation of the package's source files.

6.3 OpenACS Package Manager (APM)

The Open Architecture Community System Package Manager (APM), now Red Hat's Content and Collaboration Management Community Platform [18], serves as the best system for comparison with the iLearn package management system as they share similar design goals. The major difference is that the iLearn package management system added an emphasis on the design of a package repository to be integrated in to the entire system through web services. OpenACS, like each of the previously mentioned systems with dynamic organization of modular components, was developed before the increased popularity of web services. This thesis not only designs and implements a package system based on previous work in OpenACS, but introduces the beneficial enhancements of applying web services to such a system. By allowing communication between the package manager, repository and discussion forum through the use of web services, the package management system no longer needs multiple disjoint locations for relevant information. This new improvement is not merely a convenience for site administrators and developers, but offers a system capable of easily gathering user feedback and comments and quickly distributing the knowledge to a variety of audiences. This follows the open source movement's basic idea of allowing source

code to be easily read and redistributed, and evolving the software based on continual improvement.

Chapter 7

Future Work

As with any first effort towards software development, there are still improvements that can be made in the system left for future work. Here the focus of future development is on the web services and package repository, but also discusses considerations to improve the package manager that have been omitted of this current version in the interest of time.

7.1 Web Services

The web services exposed by the package repository API in section 5.2 simply begin to demonstrate the usefulness of web services in a package management system. This section indicates the possible improvements to the package repository web services API in terms of caching for better performance, and extension for other useful information.

7.1.1 Caching Repository Data

The most important and imminent change to the package repository web services API is the addition and modification of methods to support caching of repository data. This will be accomplished by overloading the repository method that retrieves every package summary to accept a `DateTime` argument indicating the earliest submission

date that should be returned in the set of packages. Through this new method, the package manager can simply ask for the latest updates to the repository packages to update the local cache.

Note that the other package retrieval methods that are version specific would become less frequently used as packages start relying on their cached data instead of constantly retrieving real-time data. Using cached data would improve performance and protect against the possibility of an unreachable repository, but may be relying on stale data. In this system, the package data is unlikely to change frequently and daily updates should be more than sufficient for a reasonably current cache.

The strategy for generating the cache of repository data for packages is equally applicable to package comments. In a future revision of the package manager and repository, two changes will be made to the API in relation to the web methods for comment retrieval. In addition to simply overloading the method for getting comments by accepting an argument the earliest timestamp, it would make more sense to also retrieve all the comments for every package posted after a certain timestamp. Retrieving all the comments after a certain timestamp allows the package manager to build a complete cache of the repository data for all the comments as well as the package summaries.

Another difference from generating a cache of package information is that comments will also be more frequently posted than the submission of new packages. Therefore in this case, synchronizing the local cache with the repository data on a daily basis will be insufficient for displaying recent posts about a package. The method for getting comments must accept an argument for the earliest timestamp so that the local package manager can retrieve any of the latest comments if the repository is available, while still relying on the cached data for performance optimization. This approach is advantageous to the current implementation which simply retrieves all the comments since the amount of data sent as XML through the web service will be minimized for packages containing several comments. This also allows at least a partial listing of comments if and when the repository is unreachable.

7.1.2 Multiple Repositories

Currently the assumption in the package management system is that only one repository exists at a known location. As the popularity of the package management system grows, it is possible that multiple repositories may come in to existence. To scale the local system to accommodate the possibility of multiple repositories, it becomes more important to implement the caching mechanism so that the package manager can periodically collect information from the multiple known repositories. This prevents the system from limiting itself to data from a single repository, and also maintains acceptable performance by avoiding the invocation of a web service for each repository when querying for a large list of packages.

The issue of synchronizing package comments becomes rather difficult now that repositories must either maintain their own comments, or act as peer distributed systems sharing the same comments. While the simple implementation is a benefit of having repositories managing their own comments, it violates the previously stated principle of creating a united community by instead creating several disjoint locations for related information. The distributed commenting system approach is rather difficult to implement and requires package repositories to have knowledge of each other. Furthermore, merging a collection of package comments would be a difficult task to accomplish programmatically and may require human intervention to be feasibly implemented.

The recommendation here is to avoid attempting to distribute user submitted data between multiple locations. If the need for multiple repositories arises, the additional repositories should only maintain a collection of packages and retrieve or post user submitted data to a single primary repository. The primary repository, in addition to collecting package files, has the added responsibility of maintaining the package comments submitted by users of the system. Thus the additional repositories use the package comment caching mechanism described in section 7.1.1 the same way a package manager would.

Site administrators consuming web services from multiple repositories for package

files will thus only need one repository for comments. The primary repository guarantees to have the latest information while the additional repositories may potentially have outdated cached data. Therefore, the additional repositories are sufficient for package managers updating local caches of package comments, but the primary repository is the best source for receiving the latest comments. Alternatively, depending on experimental usage data, the additional repositories may guarantee sufficiently current data by updating their caches frequently enough so that package managers may receive current data from any known repository in a scheme to distribute the demand amongst the repositories more evenly instead of potentially overloading the primary repository.

7.1.3 Additional Functionality

In addition to improving performance, the web services may be modified to supply a greater wealth of information useful for site administrators. One example would be to introduce a ratings system as a metric for how useful or popular a package is. The package repository may also record usage information, either from user navigation on the site or based on information sent by the package manager through web services. Possible relevant information would be linking packages that are often used together, or linking packages that are good substitutes.

7.2 Package Manager

Although the package manager will require modifications to make use of some of the new web services considered in section 7.1, this section describes other changes that would improve the local package management system.

7.2.1 Locking Packages

An important point neglected in the implementation of the package management system is the locking of packages during operations such as installation, upgrade and

removal. Because of the dynamic nature of the system, and the design goal for the mentioned operations to be simple and minimally intrusive on the site, a package may be asked to handle a request while the package manger attempts to modify the package. Currently the request will simply receive a 404 Not Found Error, where a more user friendly solution is appropriate. By locking packages, the user making a request could be redirected to a friendlier error message indicating that the particular request for certain functionality is either under maintenance or is in the process of being removed from the site.

This change can be implemented through the path resolution module explained in section 4.2.8. In preparation for package modification or removal, the package manager would register the package to be locked through the path resolution module. Until the lock is removed, the path resolution would appropriately rewrite the path to a standard page indicating the reason for the current request to be denied. Once the package modification is complete, the package references will be removed in the case of uninstalling the package, or the package operation will return to a working and upgraded state.

7.2.2 Platform Independence

The design goals mentioned the desire to reach a broad audience through support for multiple databases and platforms. While the current implementation handles support for many database systems, the only supported operating system is the Microsoft Windows platform since installation of the .NET Framework is required to compile and run the package management system. Previously described in section 3.2.2, support for various flavors of UNIX will be considered as the Rotor [20] and Mono [22] projects mature and allow the .NET Runtime to function on other operating systems.

Chapter 8

Conclusions

In conclusion, the package management system meets the design goals set forth to create a framework benefiting web site administrators and developers. Modeled after package management systems in operating systems, EMACS and OpenACS, the design described in this thesis has selectively implemented useful features for the purpose of running a web based application.

The iLearn system introduces a package repository with a web service interface creating several new possibilities for package management. Users of the package management system can not only obtain packages programmatically, but they can also receive current information about the latest releases and dynamically resolve any dependency issues during install and upgrade operations. This eliminates the need for site administrators to constantly monitor repositories and message boards, traditionally at separate locations, for news about recently developed packages because instead the desired information is included in the package management interface of the administered site. The ability to resolve dependencies by dynamically querying the repository to obtain package dependencies dramatically increases the usability of the package system. Site administrators will no longer have to search for necessary components and gather them for separate installations in order to add functionality with multiple dependencies.

Furthermore, another advancement in package management introduced through web services is the ability to participate in a community simply by using the system.

Package developers and site administrators have the added advantage of contributing and exploring the knowledge of their peers through the use of comments and feedback concerning packages. Future advances can expand the web services interface to expose more useful information, such as creating a ratings system for packages and users. All this useful information ultimately benefits the developers as they easily receive large amounts of feedback about their work, which results in the release of better packages for site administrators to incorporate to their sites.

The iLearn package system demonstrates the power and usefulness of web services and effectively integrates them with an existing concept in to a new system to simplify the world of web site administration and development.

8.1 Obtaining the Code

To obtain the code through CVS, execute the following commands:

```
cvs -d :pserver:anonymous@agrumet.mit.edu:/cvsroot login
(Hit ENTER when prompted for password.)
cvs -d :pserver:anonymous@agrumet.mit.edu:/cvsroot co ilearn
```

Appendix A

Package Configuration File

```
<?xml encoding=''US-ASCII''?>

<!ELEMENT Package (Properties, Files, Portlets, Dependencies,
    Settings, References)>

<!ELEMENT Properties (Name, Description, Version, Contact, Singleton,
    PackageClassName, Comment, ReleaseDate)>
<!ELEMENT Name EMPTY #REQUIRED>
<!ATTLIST Name Value (string) #REQUIRED>
<!ELEMENT Description (#PCDATA)>
<!ELEMENT Version (Value, Serial, #PCDATA) #REQUIRED>
<!ATTLIST Version Value (#PCDATA) #REQUIRED>
<!ATTLIST Version Serial (#PCDATA) #REQUIRED>
<!ELEMENT Contact (Name, Email, Url)>
<!ATTLIST Contact Name (#PCDATA)>
<!ATTLIST Contact Email (#PCDATA)>
<!ATTLIST Contact Url (#PCDATA)>
<!ELEMENT Singleton (Value)>
<!ATTLIST Singleton Value (true | false)>
<!ELEMENT PackageClassName (Value) #REQUIRED>
```

```
<!ATTLIST PackageClassName Value (#PCDATA)>
<!ELEMENT ReleaseDate (Value)>
<!ATTLIST ReleaseDate Value (#PCDATA)>

<!ELEMENT Files (File*)>
<!ELEMENT File (type, name)>
<!ATTLIST File Type (source | standalone | database | devenv | nant)>
<!ATTLIST File Name (#PCDATA) #REQUIRED>

<!ELEMENT Portlets (Portlet*)>
<!ELEMENT Portlet (Name, File, #PCDATA)>
<!ATTLIST Portlet Name (#PCDATA) #REQUIRED>
<!ATTLIST Portlet File (#PCDATA) #REQUIRED>

<!ELEMENT Dependencies (Dependency*)>
<!ELEMENT Dependency (Package, Version, Type)>
<!ATTLIST Dependency Package (#PCDATA) #REQUIRED>
<!ATTLIST Dependency Version (#PCDATA)>
<!ATTLIST Dependency Type (required | conflicts) #REQUIRED>

<!ELEMENT Settings (Setting*)>
<!ELEMENT Setting (Name, Default-Value, Global, #PCDATA)>
<!ATTLIST Setting Name (#PCDATA) #REQUIRED>
<!ATTLIST Setting Default-Value (#PCDATA) #REQUIRED>
<!ATTLIST Setting Global (true | false)>

<!ELEMENT References (Reference*)>
<!ELEMENT Reference (Name)>
<!ELEMENT Reference Name (#PCDATA) #REQUIRED>
```

Bibliography

[1] Edward C. Bailey. *Maximum RPM.* Red Hat Software, Inc., Triange Park, NC, 1997.

[2] Jonathan E. Cook and Jeffrey A. Dage. Highly Reliable Upgrading of Components. Technical report, New Mexico State University, Department of Computer Science, 1998.

[3] The Debian GNU/LINUX FAQ. http://www.debian.org/doc/FAQ.

[4] FreshPorts - the place for ports. http://www.freshports.org/.

[5] Richard S. Hall, Dennis Heimbigner, André van der Hoek, and Alexander L. Wolf. The Software Dock: A Distributed, Agent-based Software Deployment System. In *The Proceedings of the 1997 International Conference on Distributed Computing Systems.* University of Colorado at Boulder, Computer Science Department, 1997.

[6] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. Technical report, University of Colorado at Boulder, Computer Science Department, 1999.

[7] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. Specifying the Deployable Software Description Format in XML. Technical report, University of Colorado at Boulder Computer Science Department, 1999.

[8] Scott Henninger. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. Technical report, University of Nebraska-Lincoln, Department of Computer Science and Engineering, 1999.

[9] iCampus Home. http://icampus.lcs.mit.edu/.

[10] Microsoft .NET Framework Software Development Kit. http://msdn.microsoft.com/downloads/list/netdevframework.asp.

[11] Ethan L. Miller, Kennedy Akala, and Jeffrey K. Hollingsworth. Using Content-Derived Names for Package Management in Tcl. In *6th Annual Tcl/Tk Conference*, pages 171–179. University of Maryland Baltimore County and University of Maryland, Computer Science Departments, 1998.

[12] NAnt: A .NET Build Tool. http://nant.sourceforge.net.

[13] Steve Price. FreeBSD Release Engineering for Third Party Software Packages. Technical report, The FreeBSD Documentation Project, 2002.

[14] The FreeBSD Documentation Project. *The FreeBSD Handbook.* The FreeBSD Documentation Project, 1999.

[15] The FreeBSD Documentation Project. *FreeBSD Porter's Handbook.* The FreeBSD Documentation Project, 1999.

[16] Bryan Quinn. OpenACS 4.5 Package Manager Design. Technical report, OpenACS Community, 2000. http://openacs.org/doc/openacs-4/apm-design.html.

[17] Bryan Quinn and Todd Nightingale. OpenACS 4.5 Package Manager Requirements. Technical report, OpenACS Community, 2000. http://openacs.org/doc/openacs-4/apm-requirements.html.

[18] Red Hat's Content and Collaboration Management Community Platform. http://www.redhat.com/software/ccm/community/.

[19] Richard M. Stallman. EMACS: The Extensible, Customizable Self-Documenting Display Editor. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Lab, 1984.

[20] David Stutz. The Microsoft Shared Source CLI Implementation. Technical report, Microsoft Corporation, 2002. http://msdn.microsoft.com/library/en-us/dndotnet/html/mssharsourcecli.asp.

[21] The Red Hat Development Team. *The Official Red Hat Linux 4.2. User's Guide.* Red Hat Software, Inc., Triange Park, NC, 1997.

[22] Ximian Mono Project. http://www.go-mono.com/.

[23] Yunwen Ye and Fischer Gerhard. Promoting Reuse with Active Reuse Repository Systems. In *Proceedings of the 6th International Conference on Software Reuse.* University of Colorado at Boulder, Computer Science Department, 2000.